14

DIC 06



panorama

RETROEUSKAL

z88dk

LA LIBRERÍA SPRITE PACK (II)

analisis

SLOWGLASS

ensamblador

ENSAMBLADOR DEL Z80 (I)

hardware

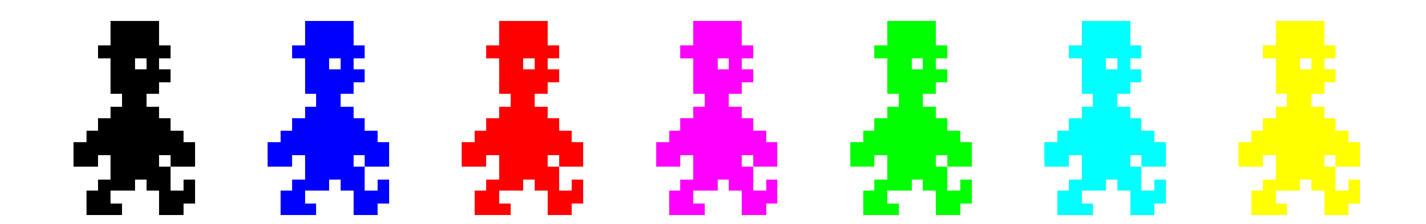
DIVIDE INTERFACE IDE ATA 16 BITS PARA SINCLAIR ZX SPECTRUM Y COMPATIBLES

input

ENTREVISTA A ALFONSO FERNÁNDEZ BORRO "BORROCOP"

juanje

http://magazinezx.speccy.org



Ha transcurrido prácticamente un año desde que el último número de la revista salió a la luz. Un año es mucho tiempo pero, echando la vista atrás, da la impresión de que el tiempo ha pasado volando. Un intervalo de tiempo demasiado grande, hasta para una publicación que no tiene fecha de entrega.

Lamentablemente, los responsables y colaboradores cada vez encontramos menos huecos para dedicarlos a esta tarea. Tenemos que reconocer que casi todos los artículos llevan meses maquetados, pero ha costado un mundo cerrar el número en sí. Por tanto, es momento de pararse a pensar en cuál va a ser el futuro de MagazineZX. Habrá que establecer un período de reflexión y estudiar qué camino tomamos. Lo que está claro es que, de una manera u otra, buscaremos una vía de expresión para los contenidos que compartimos con vosotros en esta revista.

Centrándonos ya en el número que nos ocupa, comenzamos con un resumen de la pasada Retroeuskal, una reunión que, año tras año, va ganando tanto en calidad como en visitantes. Javier Vispe ha sido el encargado de narrar la crónica.

En nuestra habitual sección de análisis, comentamos Slowglass, otro juego rescatado del fondo de un cajón gracias a las gestiones de Juan Pablo López-Grao, webmaster de SPA2.

En el apartado dedicado al hardware, Julio Medina nos presenta el divIDE, un dispositivo que hará las delicias de todos los que seguimos prefiriendo la máquina real al uso de emuladores.

Tampoco podían faltar a la cita las clásicas secciones de programación en Z88DK, Ensamblador y la entrevista. En esta ocasión, el personaje es Alfonso Fernández Borro, grafista que trabajó con casi todas las compañías de la época dorada de los 8 bits en España y que actualmente colabora con CEZ en la creación de nuevos juegos y remakes.

Esperamos, como es lógico, que encontréis interesante el contenido de este número. Y aprovechamos para recordar que la dirección de correo electrónico de contacto está abierta a todo tipo de mensajes, sugerencias y colaboraciones que queráis enviarnos.

Redacción de MAGAZINE ZX





Año IV nº14 Dic 06

Redacción

Miguel A. García Prada
DEVIL_NET

Pablo Suau
SIEW

Federico Álvarez
FALVAREZ

Ilustración de Portada

Juanje Gómez
DEV

Colaboraciones en este número

Santiago Romero
NOP

Josetxu Malanda
HORACE

Javier Vispe
ZYLOJ

Julio Medina

Maquetación en PDF

Javier Vispe
ZYLOJ

Contacto
magazinezx@gmail.com

índice

panorama

RETROEUSKAL '06

A finales del pasado mes de Julio se celebró Retroeuskal dentro de la Euskal Encounter, la "party" con más solera de nuestro país. Desde hace tres años, Bilbao también es una cita ineludible para los seguidores de los sistemas vintage.

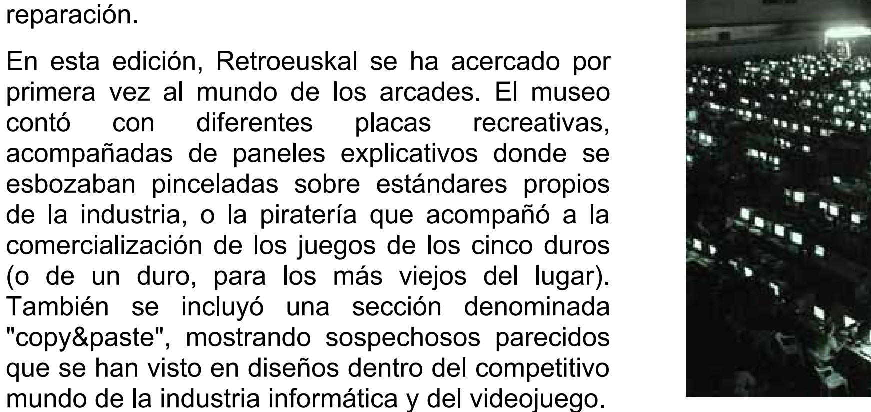
Bajo la sabia batuta de Iñaki Grao, y acompañado de un voluntarioso grupo de colaboradores, Retroeuskal continúa con su ritmo ascendente. Lo que fue un encuentro espontáneo entre usuarios de ordenadores de 8 y 16 bits, ahora es una de las secciones oficiales de la Euskal. Durante cuatro días, los asistentes a la party pueden degustar múltiples contenidos retro.



Cartel de Retroeuskal

Retroeuskal funciona gracias al esfuerzo de aficionados con ganas de dar a conocer la historia y la actualidad de máquinas "obsoletas", bien sea con material propio, apoyo logístico, mano de obra, ponencias, etc. Todo el programa que ofrece la reunión es fruto de la aportación desinteresada de gente que disfruta colaborando en la organización.

"¿QUÉ TIENE RETROEUSKAL, QUE A TODO EL MUNDO LE MOLA?"



La propuesta de funcionamiento de la reunión gira en torno a las actividades de sus diferentes zonas: museo, talleres, conferencia/mesa redonda, demoshow 8 bits, cena interplataformas y taller de reparación.

En esta edición, Retroeuskal se ha acercado por primera vez al mundo de los arcades. El museo contó con diferentes placas recreativas, acompañadas de paneles explicativos donde se esbozaban pinceladas sobre estándares propios de la industria, o la piratería que acompañó a la comercialización de los juegos de los cinco duros (o de un duro, para los más viejos del lugar). También se incluyó una sección denominada "copy&paste", mostrando sospechosos parecidos que se han visto en diseños dentro del competitivo mundo de la industria informática y del videojuego.

Por último, el museo se completó con la zona denominada juegódromo. Varios espacios habilitados con ordenadores de 8 y 16 bits permitían a los visitantes disfrutar de juegos clásicos, como si fuera un salón recreativo de los 80, pero con máquinas en modo "free play". Esta iniciativa tuvo bastante éxito, y se convirtió en un punto de atracción, incluso para las nuevas generaciones que no tuvieron el placer de asistir al nacimiento del mercado de los videojuegos en nuestro país.

¿MSX Y CPC EN UNA REVISTA DE SPECTRUM?

Los talleres, en esta ocasión, tuvieron como protagonistas destacados al Amstrad CPC y el MSX. Ambos ordenadores pueden ejecutar Symbos, un sistema operativo de ventanas multitarea que explota las capacidades de estas máquinas hasta límites jamás soñados. Su creador, Jörn Mika (Prodatron), mostró en persona su funcionamiento, y explicó como incluso las aplicaciones trabajan directamente en ambas plataformas sin


The Matrix has you...

modificación alguna. También reveló la "relativa" facilidad que tiene Symbos para ser adaptado a otros sistemas con un Z80 en sus entrañas. Obviamente, esta cuestión interesó a los usuarios de máquinas Sinclair allí presentes.


Jörn con su Symbos en acción

Néstor Soriano, también conocido como Konamiman, o "el Líder", dentro de la comunidad MSX, mostró en otro de los talleres el proyecto DUMAS. Bajo esta críptica denominación se esconde una expansión de 512K de memoria, una interfaz USB para poder usar cualquier dispositivo con esta conexión (USBorne) y una tarjeta ethernet. Los asistentes pudieron observar en vivo su estado de desarrollo y funcionamiento, tanto con pendrives como con utilidades de telnet y chat programadas por usuarios ajenos al proyecto.

Por último, no se puede dejar sin citar el taller de fabricación de arcade sticks. Crowvarela hizo un completo repaso a los componentes disponibles comercialmente para montar un mando a nuestro gusto, ofreciendo las recomendaciones necesarias para respetar al máximo una buena ergonomía y obtener el máximo rendimiento. Su exposición fue acompañada por fichas explicativas y material para poder comparar in situ.


Retro-rangers attack

NO SE VAYAN TODAVÍA, AÚN HAY MÁS

La conferencia de esta edición llevaba por título "¿Horizontes lejanos?: Explorando el límite hardware de las máquinas clásicas". Bajo el formato de mesa redonda que tan buenos resultados ha dado en anteriores ocasiones, Jörn Mika y Néstor Soriano, hicieron un repaso a su implicación en los nuevos desarrollos para sistemas obsoletos. Los conferenciantes expusieron motivaciones, formas de trabajar, contacto con la comunidad, etc. Todo esto aderezado y salpicado con las preguntas de los asistentes. La anécdota fue la necesidad de contar con traducción simultánea por la presencia de Prodatron, lo que no impidió un ágil desarrollo de la charla.

Otra de las novedades de este año fue la implantación de un taller de reparación, gracias a la iniciativa de Jaime Lapeña (Rockriver). En la anterior edición ya se dedicó espontáneamente tiempo a la recuperación de máquinas estropeadas, y este año se optó por oficializar la actividad como un servicio más a la comunidad de retro-adictos. En muchas ocasiones, los problemas que sufren estos aparatos tienen fácil solución teniendo la información adecuada y el material necesario. Por ello, también se ofrecieron consejos para todo aquel que se pasó por este "hospital de campaña".


Rockriver y Z80user en el taller de reparación

Fiel a su cita, "La hora de los 8 bits" volvió a tomar las pantallas gigantes de Euskal Encounter, con un demoshow de las últimas producciones de la scene internacional. Una reunión que en sus inicios surgió desde y para la demoscene no puede dejar pasar la oportunidad de contar con las piruetas y "más difíciles todavía" que siguen saliendo de los circuitos ochenteros de Commodore, CPC, MSX, Spectrum... Sería magnífico que en futuras ediciones se pudiera llegar a contar con producciones exclusivas para esta sección. Pero esto queda a expensas de lo que dicte el futuro...

Por último, nos queda hablar de la cena de hermanamiento interplataformas. Lo que en su momento fue motivo de rivalidades infantiles, en Retroeuskal se toma a guasa y se aprovecha como excusa para seguir charlando afablemente sentados en la mesa. No importa que seas de

MSX, de Atari o de Sinclair. Importa que conozcas a gente y aportes tu granito de arena para disfrutar de un largo fin de semana. Aunque no podemos negarlo, la cabra siempre tira al monte, y no se puede evitar echar en cara esos píxeles tan gordos o esa paleta de colores tan escasa del ordenador del comensal de enfrente. ;)



Una chica jugando al Spectrum. En Retroeuskal, impossible is nothing

BUENO, ¿Y QUÉ HAY DEL SPECTRUM?

Por supuesto, las máquinas Sinclair tuvieron su porción de protagonismo en Retroeuskal.

Uno de los invitados de esta edición en la zona de proyectos, fue Mhoogle, el buscador de contenidos de Microhobby. Como bien sabemos, la criatura de Josetxu Malanda (Horace), permite el acceso online a cualquier artículo de la revista. Esta herramienta se ha convertido en imprescindible para aquellos que necesitan acceder a la hemeroteca en busca de contenidos sobre Spectrum de forma casi instantánea.



Desde el Este con amor

En la sección de copy&paste se mostró el porqué de la similitud entre el Jupiter ace y el ZX81 (compartir equipo de diseño tiene estas cosas...). También estuvieron presentes varios clones de Spectrum fabricados en la Europa del este. Nunca dejará de sorprender la increíble cantidad de versiones que existen de esta familia de ordenadores.

El juegódromo incluyó una zona exclusiva dedicada a máquinas Sinclair. Se pudieron observar y manipular prácticamente todos los ordenadores de la marca, desde un ZX81 hasta un Plus 3. Además, se contó con la posibilidad de usar joysticks y una pistola óptica para ofrecer la experiencia retrojugona más completa posible. Sobre este aspecto, hay que comentar que una de las mayores virtudes de Retroeuskal es poder observar los sistemas originales en funcionamiento, sin tener que recurrir a emuladores.

Los Spectrum permitieron probar más de un centenar de juegos, gracias a los nuevos desarrollos de hardware de José Leandro Novellón. El supercartucho para Interface II dejaba a disposición de los visitantes 15 juegos que, gracias a este soporte, se podían cargar en memoria de forma casi instantánea. De la misma forma, dos interfaces de tarjetas Compact Flash conectados a un Plus 2e y un Plus 3e respectivamente, daban acceso a una amplia lista de títulos a través del menú zxcfgui. Esta combinación de hardware y software permitió ofrecer una pequeña parte del catálogo de Spectrum al público general. Los usuarios podían elegir el juego seleccionándolo en la lista y disfrutarlo sin tener que sufrir la desesperante sensación de cómo hacer funcionar un equipo que no se conoce. En definitiva, se consiguió acercar estos ordenadores a los neófitos en la materia, evitando en gran medida la barrera de un interfaz poco amigable para los estándares actuales.



Retroeuskal es para todos los públicos

A la espera de que empiece a moverse de nuevo la maquinaria de la próxima Retroeuskal, podemos hablar de un evento con ganas de crecer, perfeccionarse y mostrar que hubo vida antes de la era PC. Como ya se ha comentado, su futuro dependerá de la implicación de los aficionados a este mundo "tecno-arqueológico" al que pertenecemos. Pero si uno lo piensa detenidamente, ¿por qué no aprovechar esas fechas veraniegas para pasar un fin de semana compartiendo tus gustos con gente, y disfrutando de la oferta turística que ofrece el País Vasco? Si te animas, nos vemos el 20 de Julio de 2007.

LINKS

http://www.retroeuskal.org

JAVIER VISPE

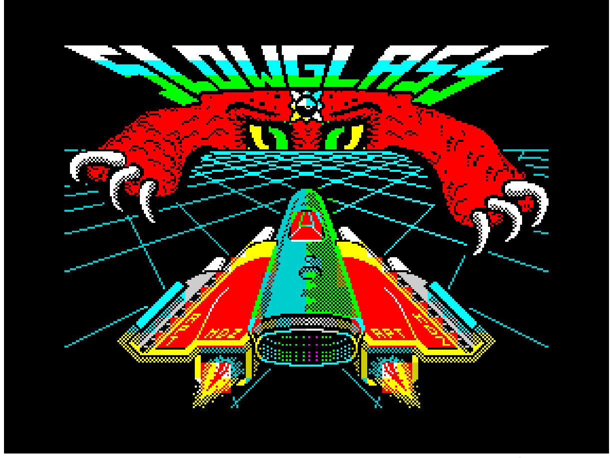


En este número analizamos pura producción nacional: Slowglass, de la mano de Javier Vispe.

SLOWGLASS

Título	Slowglass
Género	Matamarcianos
Año	1990
Máquina	48K/128K
Jugadores	1 Jugador
Compañía	Inédito
Autor	Manuel Domínguez, Alberto Pérez
Otros comentarios	N/A

2005 nos trajo un buen número de juegos inéditos que no se comercializaron en su momento. Uno de ellos es Slowglass, el cual disfrutamos gracias a la colaboración de Nach con Juan Pablo López-Grao para localizar a sus autores, Manuel Dominguez Zaragoza y Alberto Pérez Torres. Su único trabajo público conocido hasta ahora era Cyberbig, editado en 1989 por Animagic. Esta empresa iba a comercializar el título que nos ocupa, pero tras su cierre, y fallidas negociaciones con otras distribuidoras, hizo que se quedara guardado en un cajón.

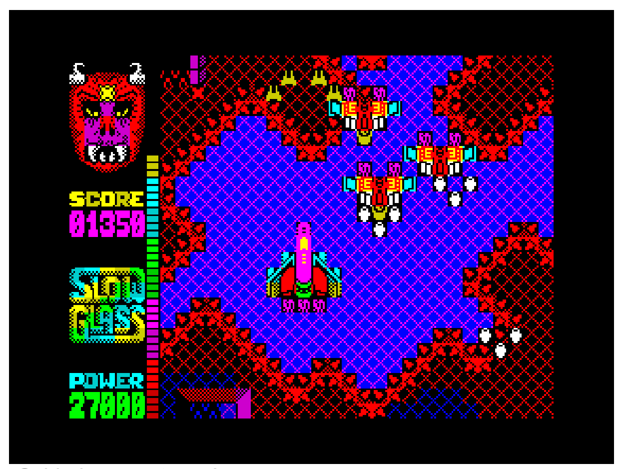


Pantalla de carga de Slowglass

Slowglass es un matamarcianos de desarrollo vertical, con la peculiaridad de que el "scroll" no es continuo, sino que avanza conforme nosotros nos desplazamos. Debemos arrastrar una nave nodriza a lo largo del escenario, para lo cual disponemos de un contador de combustible que disminuye paulatinamente mientras la llevamos. Cuando se agotan nuestras reservas, hemos de buscar unos portales con forma de cabeza monstruosa. Por allí se accede a una dimensión paralela, reflejo de la primera. En ella nos desplazaremos hacia los lagos de cristal, donde se recarga el depósito, y después regresaremos de nuevo por los portales para continuar con nuestro cometido. El efecto espejo que relaciona ambas dimensiones, supone que nuestro avance en una de ellas, implica ir hacia atrás en la otra. Por tanto, al regresar de un "plano de realidad" a otro, habremos desandado parte del camino que habíamos recorrido.

La misión que se nos encomienda en el juego se verá dificultada por las naves enemigas que nos atacan. Cada vez que recibimos un impacto, disminuye nuestra barra de energía. Sin embargo, son mucho más peligrosos los proyectiles que surgen de vez en cuando en pantalla. Estos nos quitarán una vida con su contacto, al igual que los laterales de los portales dimensionales, o las estructuras que aparecen cerca del final de nuestro recorrido. Es recomendable movernos con

cautela cerca de dichas zonas.



¡Cuidado, nos atacan!

Para facilitar la misión, contamos con aprovechar los objetos que aparecen con movimiento diagonal, y que ofrecen invulnerabilidad o una mayor potencia de disparo durante un período de tiempo limitado.

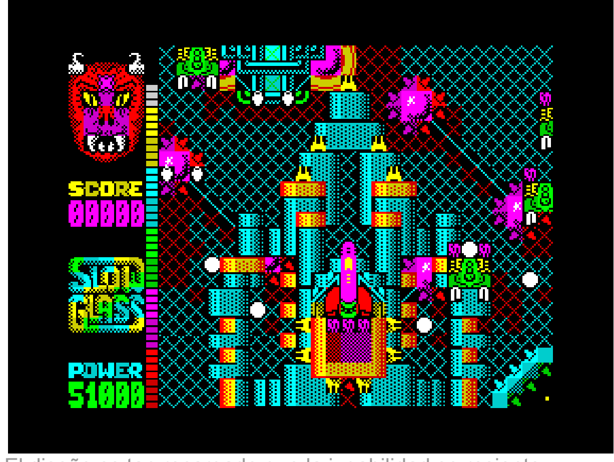
Si conseguimos llegar con la nave nodriza hasta el final del trayecto, todavía nos esperan emociones fuertes (y algo surrealistas, porque no decirlo), y que dejo en incógnita para aquellos que se propongan terminarlo.

Slowglass destaca sobre todo por el trabajo realizado en su apartado visual. Hace uso intensivo del color, tanto en los escenarios como en los sprites. El "scroll" de desplazamiento se produce de 8 en 8 píxeles para evitar el problema de la mezcla de atributos, pero está resuelto de forma muy fluida. Además, cuenta con varios planos de profundidad. Sin embargo, este derroche de barroquismo termina convirtiéndose en un inconveniente para la jugabilidad:

- Nuestra nave es muy grande en comparación a la zona de juego, por lo que se hace difícil esquivar las ráfagas de disparo enemigas.
- Los fondos tan recargados y coloridos confunden a la hora de ver los misiles o los elementos del decorado que nos quitan una vida.
- La activación del "scroll" para avanzar se realiza cuando nuestra nave se sitúa en la parte superior de la pantalla. Si por una casualidad, en ese momento sale un misil por esa zona, no tenemos tiempo de reacción para esquivarlo.

Estos fallos son una fuente de frustración para el jugador, por ese factor de suerte que no puede controlar por medio de su habilidad. Son muertes que podrían haberse evitado para que Slowglass no se viese perjudicado en su disfrute. No es un caso aislado, ya que muchos juegos de la época no tuvieron la fortuna de contar con una estructura de producción y testeo tan profesionalizadas como las actuales. De todas formas, también debemos preguntarnos si hace 15 años, habríamos sido más condescendientes con estos factores, cuando eramos más inocentes y no llevábamos tantas horas de vuelo en esto de los videojuegos.

Viendo la evolución del trabajo de Manuel y Alberto, es una lástima que no pudieran disponer de más medios y tiempo que dedicar al Spectrum. Seguramente habríamos visto cosas muy interesantes en nuestras pantallas.



El diseño es tan recargado que la jugabilidad se resiente

descárgalo de:

WOS

LINKS

http://spa2.speccy.org/spanish/adventure.htm

http://www.arrakis.es/~caad/ficheros/spectrum.html

Valoraciones

originalidad	[7]	jugabilidad	[6]
gráficos	[8]	adicción	[6]
sonido	[6]	dificultad	[7]

JAVIER VISPE

DIVIDE.
Interface IDE ATA 16 bits para SINCLAIR ZX SPECTRUM y compatibles

Año 2002

Autor Pavel Cimbal. "ZiloGator"

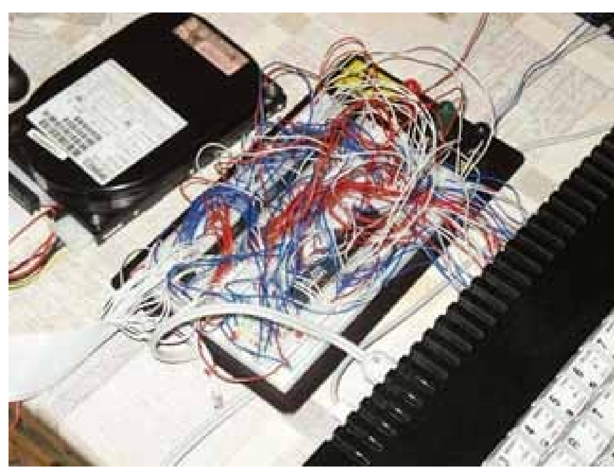
País Chequia



DESCRIPCIÓN DEL INTERFACE

Si hacemos una revisión histórica de interfaces no vamos a encontrar nada parecido, Dan Anthoi con su "expansion system" o Ans Summer, Pera Putnik en su interesante pagina de hardware nos propone dos modelos, en modo 8 ó 16 bits, de él derivó el +3e de Garry Lancaster y en la "scene" rusa también existen interfaces de disco duro con uso de zx-bus.

divIDE se diseñó en 2002 a petición de un grupo de usuarios y tras 15 días de pruebas se realizó su diseño profesional, todo un reto.



En la imagen los componentes en una protoboard y disco duro Conner

Una breve descripción del aparato nos dice que su tamaño es similar a una placa de PlusD y realmente es muy superior en prestaciones. Usa pocos componentes y modernos. No emplea integrados "difíciles" como el i8255 y su elaboración es sencilla a partir de placa profesional. Cuenta con una eeprom de 8Kb, una RAM de 32Kb, 3 GAL y 3 integrados de la serie 74HC**, 3 transistores... No hay conexión Centronics.

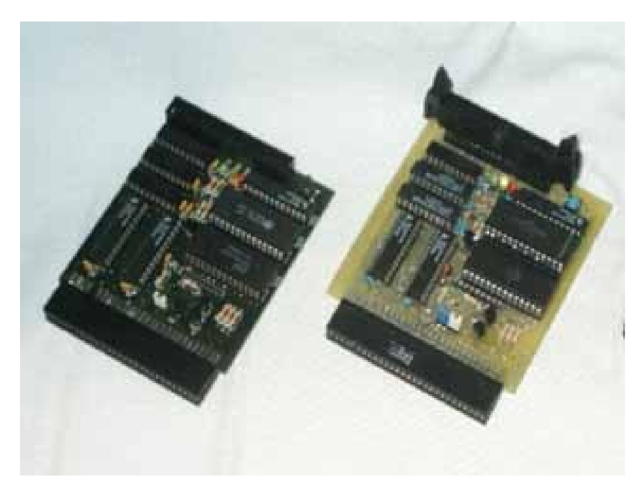


El divide en acción

Se ha dicho que las GAL ya no se fabrican. Esto es falso, se siguen desarrollando y tanto las originales elaboradas por Lattice® como las Atmel® u otros fabricantes son perfectamente válidas, sólo que hay que configurarlas con un programador especifico que las soporte.

Existe la posibilidad de realizar la placa de manera casera. Realmente no lo recomiendo, pues cuenta con múltiples vías que debe ser mecanizadas con broca de 0.5.

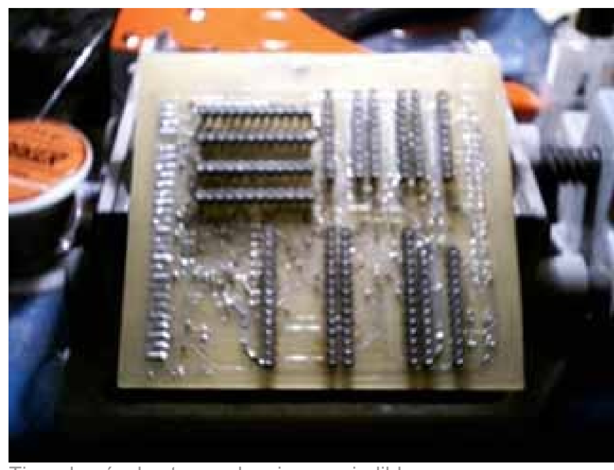
Otra característica que lo separa del resto de interfaces, es ser un producto en el que el desarrollador no obtiene beneficio económico. Se adquiere por 20€ contactando con el autor más los gastos de envío. Por correo en el kit se incluye todo lo necesario: GAL programadas, integrados, la placa y algo realmente difícil de encontrar, el conector del bus del Spectrum.



Yo no tengo manera de metalizar las placas de forma casera, tuve que añadir un conector para faja IDE al propio conector normal. Mas que un divIDE parece un Mihura ;D



Diversos tipos de conectores desoldados



Tiras de zócalos torneados, imprescindibles

¿QUÉ LO HACE REALMENTE INTERESANTE?

Funciona con todos los modelos de Spectrum, desde el Sinclair 16KB hasta los modelos de AMSTRAD®, incluidos los clónicos rusos y los modelos de países del Este, didaktik m, gama, etc.

Se adapta a la RAM disponible y carga los ficheros compatibles para ese modelo.

No es necesario modificar nada al Spectrum. No se necesita cambiar la ROM pues, como ocurre con el PlusD o Beta disk, el firmware lo incorpora, el cual es "reflasheable". De hecho no sólo es un interface IDE, también es un grabador de su propia flash eeprom. No necesitamos un programador externo para configurar el interface, cargamos el software en cinta y seguimos sus instrucciones. El Spectrum queda listo para trabajar con cualquier dispositivo IDE-ATAPI que conectemos: CDROM, disco duros con FAT16 y discos duros con FAT32 en los que exista al menos una imagen ISO 9660, unidades ZIP® y, cómo no, las compact flash de cualquier tamaño, siendo mejor las de tipo 1, al ser "lentas". Sólo admite dos unidades IDE ATAPI, maestra y esclava.



No precisa de alimentación externa, la coge del propio bus del Spectrum. Como los integrados son de la serie HC, HCT, el consumo no es elevado. Esto unido a que las compact flash pueden también alimentarse a través del bus lo hacen la configuración ideal, un sistema totalmente silencioso.

Nos olvidamos de la cinta y el disquete para siempre. Una vez configurado ya no precisamos el software, cuenta con su propia "BIOS".

La tasa de transferencia de datos es excepcional, mas de 200KB/s. Supera a cualquier interface de disco 3,5".

Si conectamos discos duros el consumo se eleva y es necesaria una fuente de PC.

Existe un proyecto denominado cfIDE, es un adaptador de compact flash a bus IDE. En Internet hay muchos productos que nos ahorrarán el trabajo de soldar un bus de compact flash, terriblemente difícil de adquirir y de soldar, y a un precio elevado.

Toda la información para la elaboración profesional del interface se encuentra en la pagina del autor. Ha padecido un accidente y ha detenido el desarrollo del proyecto por unos meses. Desde aquí deseamos su pronta recuperación.



Este adaptador solo soporta una CF, existen adaptadores de 2 CF

¿QUÉ SOFTWARE HAY DISPONIBLE?

Cuando terminamos de soldar los componentes del kit disponemos de un software específico que comprueba tanto el interface como el propio Spectrum al que hemos conectado el divIDE. Es:

Transient Bios, TBIOS versión 1.4: nos muestra con detalle el tipo de Spectrum, comprueba la RAM interna del Spectrum, la eeprom externa del divIDE, hace una prueba de escritura en todos los bancos de RAM, hace un test de reloj... Es realmente difícil describir con palabras. Al final comprueba el hardware que tengamos conectado y muestra la información del firmware del fabricante del disco duro y CDROM.



Pulsa symbol shift+a y se inicia tbios

```
CLOCKING LOGIC TEST:
Hold NMI, press key: Making 50000 edges, 25314 done, release NMI
HOST RAM TEST (128KB):
Page7 16384 OK  Page6 16384 OK  Page5 16384 OK  Page4 16384 OK
Page3 16384 OK  Page2 16384 OK  Page1 16384 OK  Page0 16384 OK
MAPRAM MODE TEST:
OK, installed
DIVIDE RAM TEST:
Bank0 08192 OK  Bank1 08192 OK  Bank2 08192 OK  Bank3 08192 OK
DIVIDE EEPROM TEST:
Open EPROM jumper for testing, press key: Eeprom 00000 OK
DETECTING AT-IDE AND ATAPI DEVICES:
DEVICE 0 (MASTER): Disk
Model: QUANTUM FIREBALL ST3.2A                    Rev: A0F.0A00
SN: 853806448204                 Buffer: 41728 sectors
Current CHS: 6256/16/63            Default CHS: 6256/16/63
Current  capacity: 6306048 sectors  Multiple sectors: 16
LBA addressable sectors: 6306048  Multiple setting: 0
DEVICE 1 (SLAVE): CD-Rom
Model: HL-DT-ST GCE-8160B                         Rev: 1.02
```

Se termina el chequeo del Spectrum

Hay cuatro grandes sistemas operativos:

1. FATWARE: versión 0.12, soporta hasta 8 particiones en FAT16, no soporta FAT32, con un tamaño máximo de 8 gigas. ;D



Fatware

2. MDOS3: Teniendo en cuenta que ahí debía haber soporte para las unidades D40 (MDOS1) y D80 (MDOS2) se crea este nuevo sistema compatible y divIDE se comporta como una unidad "virtual" de hasta 4 unidades de disco. Trabaja con ficheros *.raw de imagen de disco D40 y D80, se aprovecha todo el material disponible en este formato.

3. +Divide: Sistema compatible +Gdos, de las unidades Disciple/+D. divIDE reconoce imágenes de disco de estas unidades y trabaja con ellas a bajo nivel, por lo que hemos de ser cuidadosos al realizar labores de escritura y lectura.

4. Demfir: versión 0.5b, 0.6, realmente no es sistema operativo, es un emulador de ficheros de emulador, a partir de una imagen *.ISO 9660 contenida en el disco duro o CDROM. Sólo lectura.

El Spectrum se convierte en una extraordinaria consola con miles de videojuegos al reconocer los ficheros más habituales:



Tras pulsar NMI, aparece este menú. Pulsamos "S" y ponemos en hexadecimal el sector de arranque de la imagen ISO contenida en el disco duro Quantum Fireball



Apuntando al sector de arranque divIDE detecta automáticamente la imagen madrizx06.iso



En este caso es una Sandisk® 128MB. Sector de arranque en la dirección 00000DB. Si no ponemos un nombre válido, divIDE anota la fecha y la hora del archivo *.iso

*.SNA, *.z80 y *.SCR se cargan de forma instantánea, no importa la versión de fichero generada por el emulador de PC sobre la que se hizo el fichero que el Spectrum deba cargar, las reconoce todas.

*.TAP, podemos editar el fichero y entrar desde cualquier punto, tras pulsar "R" se efectúa el RESET y tecleamos LOAD ""... Y a disfrutar.

Por ahora los sistemas son sólo lectura, en próximas actualizaciones habrá soporte de lectura/escritura.



Cuando intentamos cargar un *.z80...



Directorio principal...vamos a jue_wos


Las demos a cualquier tamaño. Esta mega-demo llamada MQM5 tiene varias partes, es espectacular. Tiene un tamaño de 640 KB y el divide no se cuelga para nada


Cuando se manejan con precisión los ciclos de reloj se consigue esto. La foto no reproduce el efecto. Por supuesto, la conexión RGB es esencial para disfrutar al máximo y apagar la luz de la habitación

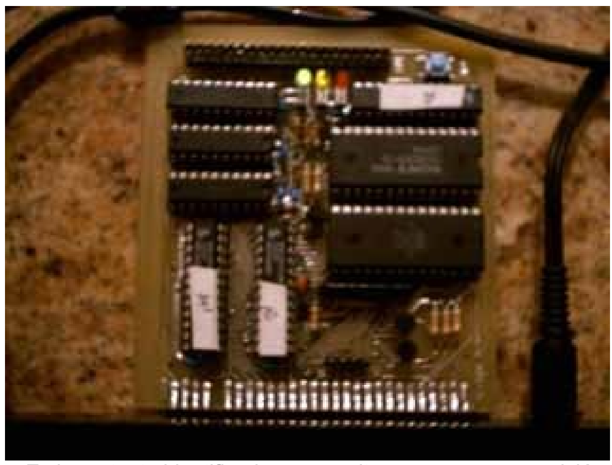
Si dentro de la imagen ISO ponemos firmware para el divIDE, podemos desde el propio interface actualizarlo con suma rapidez.

ASPECTOS TÉCNICOS DE DIVIDE

Hay dos versiones de placa, la 57b y la 57c.

La única diferencia está en los componentes discretos, se cambian los valores de varias resistencias según el tipo de transistor empleado.

Hay 4 ficheros *.JED para 3 GAL, A, M y R, los ficheros a programar son: a, m’ y r. Es importante identificarlas para evitar errores en su posición.



Es importante identificarlas para evitar errores en su posición

Los PLD GAL son matrices lógicas reprogramables. Sustituyen y "emulan" a un buen grupo de integrados lógicos con el beneficio de consumir poco.

Si partimos de kit no debe haber problema, pues, como dije, el kit incluye todo lo necesario para terminar el trabajo.

Si partimos de cero, debemos trabajar con la versión 57c, y seguir las instrucciones del autor en la pagina.

Para realizar el kit recomiendo un soldador de no más de 30W y punta fina.

Una vez terminado el trabajo y configurado nos encontramos ante el interface que nos devuelve a la actualidad a nuestro querido Spectrum y lo pone a trabajar con los dispositivos de almacenamiento modernos.

Quiero agradecer a:

Pavel Cimbal, por dar a conocer de modo altruista este interface maravilloso.

Alberto Nadal, por darme la oportunidad de reparar sus interfaces y regalarme uno con placa profesional y el adaptador de CF.

JULIO MEDINA



Página del autor

Página oficial

Demfir

MSDOS3

TBIOS

+divide

Fatware

Proyecto completo

Correo electrónico del autor



programación z88dk

LA LIBRERIA SPRITE PACK (II)

En la entrega anterior de la revista comenzamos a examinar la librería Sprite Pack y aprendimos a dibujar elementos gráficos sobre la pantalla. También aprendimos que la pantalla estaba dividida en 32x24 "celdas", cada una de las cuales podía contener un tile (cada uno de los elementos que forman parte del fondo) y/o uno o más sprites (gráficos que representaban los diferentes personajes o elementos de nuestro juego). Incluso llegamos a crear un sprite que se movía al azar por la pantalla.

En esta ocasión vamos a añadir un par de sencillos detalles más antes de comenzar a escribir nuestro propio juego; un código que luego podrá formar parte de un producto más elaborado. En concreto veremos cómo hacer reaccionar nuestro programa ante la pulsación de teclas por parte del usuario, y cómo mover un sprite utilizando este dispositivo de entrada. También aprenderemos como añadir color a los sprites. Para nuestras explicaciones haremos uso como base del código `sprite2.c` que se creó en la anterior entrega, y en el que se definía un sprite de tamaño 2x1 que se desplazaba al azar.

MOVIENDO LOS SPRITES CON EL TECLADO

Si queremos mover un sprite por la pantalla utilizando el teclado, lo primero que deberemos hacer en nuestro programa es declarar una estructura del siguiente tipo:

```
struct sp_UDK keys;
```

La variable `keys` nos permitirá asociar teclas de nuestro teclado con los diferentes controles de un joystick virtual haciendo uso de la función `sp_Lookup_Key`, tal como se puede observar a continuación:

```
keys.up = sp_LookupKey('q');
keys.down = sp_LookupKey('a');
keys.right = sp_LookupKey('p');
keys.left = sp_LookupKey('o');
keys.fire = sp_LookupKey(' ');
```

Según este ejemplo, cada vez que pulsáramos 'q' en el teclado es como si estuviéramos moviendo el joystick hacia arriba, al pulsar 'a' simulamos la dirección de abajo del joystick, y así sucesivamente. Varias de estas teclas podrían ser pulsadas simultáneamente sin ningún problema.

La función `sp_JoyKeyboard` devuelve una máscara de bits indicándonos qué controles del joystick están accionados en ese momento debido a que sus correspondientes teclas están pulsadas. La forma de interpretar el valor devuelto por esta función es igual que con cualquier máscara de bits en C, y se puede contemplar en el siguiente ejemplo, que consiste en el archivo `sprite2.c` de la anterior entrega modificado de tal forma que movamos el sprite precisamente con la combinación de teclas indicada anteriormente. Para ello realizamos movimientos relativos del sprite, cambiando cómo desplazamos el sprite en x y en y (dx y dy respectivamente) según los controles de nuestro joystick virtual que estén siendo accionados. El código nuevo respecto a la versión anterior del archivo se muestra en rojo:

```
#include <spritepack.h>
#include <stdlib.h>
#pragma output STACKPTR=61440

extern struct sp_Rect *sp_ClipStruct;
#asm
LIB SPCClipStruct
._sp_ClipStruct         defw SPCClipStruct
#endasm

extern uchar bicho1[];
extern uchar bicho2[];
extern uchar bicho3[];
struct sp_UDK keys;

uchar hash[] = {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa};

void *my_malloc(uint bytes)
{
           return sp_BlockAlloc(0);
}

void *u_malloc = my_malloc;
void *u_free = sp_FreeBlock;

main()
{
           char dx,dy,i
           struct sp_SS *spriteBicho;


            #asm
            di
            #endasm
            sp_InitIM2(0xf1f1);
            sp_CreateGenericISR(0xf1f1);
            #asm
            ei
            #endasm

           sp_TileArray(' ', hash);
           sp_Initialize(INK_WHITE | PAPER_BLACK, ' ');
           sp_Border(BLUE);
           sp_AddMemory(0, 255, 14, 0xb000);

           keys.up = sp_LookupKey('q');
           keys.down = sp_LookupKey('a');
           keys.right = sp_LookupKey('p');
           keys.left = sp_LookupKey('o');
           keys.fire = sp_LookupKey(' ');

            spriteBicho = sp_CreateSpr(sp_MASK_SPRITE, 3, bicho1, 1,
TRANSPARENT);

           sp_AddColSpr(spriteBicho, bicho2, TRANSPARENT);
           sp_AddColSpr(spriteBicho, bicho3, TRANSPARENT);
           sp_MoveSprAbs(spriteBicho, sp_ClipStruct, 0, 10, 15, 0, 0);

           while(1) {
                 sp_UpdateNow();
```

```
 i  = sp_JoyKeyboard(&keys);
                 if ((i & sp_FIRE) == 0) {
                      dx = dy = 1;
                 } else {
                      dx = dy = 8;
                 }
                 if ((i & sp_LEFT) == 0)
                      dx = -dx;
                 else if ((i & sp_RIGHT) != 0)
                      dx = 0;
                 if ((i & sp_UP) == 0)
                      dy = -dy;
                 else if ((i & sp_DOWN) != 0)
                      dy = 0;

                 sp_MoveSprRel(spriteBicho, sp_ClipStruct, 0, 0, 0, dx,
dy);
           }
}

#asm

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._bicho1
defb @00000011, @11111100
defb @00000100, @11111000
defb @00001000, @11110000
defb @00001011, @11110000
defb @00001011, @11110000
defb @00001000, @11110000
defb @00001000, @11110000
defb @00000100, @11111000

defb @00000011, @11111100
defb @00001100, @11110011
defb @00001100, @11110011
defb @00011000, @11100111
defb @00011000, @11100111
defb @01111100, @10000011
defb @01111100, @10000011
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._bicho2
defb @11100000, @00011111
defb @00010000, @00001111
defb @00001000, @00000111
defb @01101000, @00000111
```

```
defb @01101000, @00000111
defb @00001000, @00000111
defb @10001000, @00000111
defb @10010000, @00001111

defb @11100000, @00011111
defb @00011000, @11100111
defb @00011000, @11100111
defb @00001100, @11110011
defb @00001100, @11110011
defb @00111110, @11000001
defb @00111110, @11000001
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

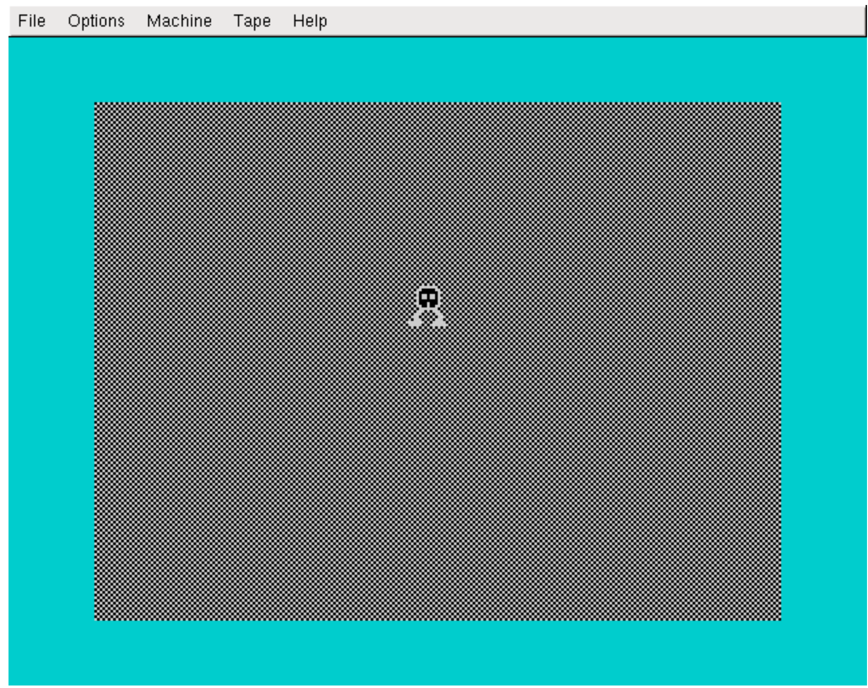
._bicho3
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

#endasm
```

Al probar el ejemplo nos habremos encontrado con un problema bastante grave: al no controlar cuándo el sprite sobrepasa los límites de la pantalla, es posible desplazar nuestro "bicho" al exterior y perder totalmente su control. Una forma de solucionarlo sería añadir un nuevo control de tal forma que al accionarlo volviéramos a colocar a nuestro bicho en el centro de la pantalla. Esto se puede conseguir realizando mapeados adicionales de teclado a los del joystick virtual. Mediante `sp_LookupKey` se puede asociar una tecla a una variable `uint`, de tal forma que más adelante, mediante el uso de otra función adicional llamada `sp_KeyPressed` podremos saber si dicha tecla está siendo pulsada en este momento.



¡Ya podemos mover a nuestro bicho con el teclado!

El código anterior se ha modificado para añadir dos nuevas funcionalidades. Al pulsar la tecla r, nuestro "bicho" volverá al centro de la pantalla. Por otra parte, la tecla b va a permitir que cambiemos el color del borde. Las modificaciones se muestran en rojo:

```
#include <spritepack.h>
#include <stdlib.h>
#pragma output STACKPTR=61440

extern struct sp_Rect *sp_ClipStruct;
#asm
LIB SPCClipStruct
._sp_ClipStruct         defw SPCClipStruct
#endasm

extern uchar bicho1[];
extern uchar bicho2[];
extern uchar bicho3[];
struct sp_UDK keys; //NUEVO (TECLAS)

uchar hash[] = {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa};

void *my_malloc(uint bytes)
{
            return sp_BlockAlloc(0);
}

void *u_malloc = my_malloc;
void *u_free = sp_FreeBlock;

main()
{
            char dx,dy,i
            struct sp_SS *spriteBicho;
            uint reset,cambioBorde;
            int borde = 1;


             #asm
             di
             #endasm
```

```
            sp_InitIM2(0xf1f1);
            sp_CreateGenericISR(0xf1f1);
            #asm
            ei
            #endasm

           sp_TileArray(' ', hash);
           sp_Initialize(INK_WHITE | PAPER_BLACK, ' ');
           sp_Border(BLUE);
           sp_AddMemory(0, 255, 14, 0xb000);

           keys.up = sp_LookupKey('q');
           keys.down = sp_LookupKey('a');
           keys.right = sp_LookupKey('p');
           keys.left = sp_LookupKey('o');
           keys.fire = sp_LookupKey(' ');
           reset = sp_LookupKey('r');
           cambioBorde = sp_LookupKey('b');

           spriteBicho = sp_CreateSpr(sp_MASK_SPRITE, 3, bicho1, 1,
TRANSPARENT);
           sp_AddColSpr(spriteBicho, bicho2, TRANSPARENT);
           sp_AddColSpr(spriteBicho, bicho3, TRANSPARENT);
           sp_MoveSprAbs(spriteBicho, sp_ClipStruct, 0, 10, 15, 0, 0);

           while(1) {
                 sp_UpdateNow();

                                 // TODO DENTRO DE ESTE WHILE ES NUEVO
(TECLADO) MENOS EL UPDATE
                 i  = sp_JoyKeyboard(&keys);
                 if ((i & sp_FIRE) == 0) {
                      dx = dy = 1;
                 } else {
                      dx = dy = 8;
                 }
                 if ((i & sp_LEFT) == 0)
                      dx = -dx;
                 else if ((i & sp_RIGHT) != 0)
                      dx = 0;
                 if ((i & sp_UP) == 0)
                      dy = -dy;
                 else if ((i & sp_DOWN) != 0)
                      dy = 0;
                 if (sp_KeyPressed(reset))
                     sp_MoveSprAbs(spriteBicho, sp_ClipStruct, 0, 10, 15,
0, 0);
                 else
                     sp_MoveSprRel(spriteBicho, sp_ClipStruct, 0, 0, 0,
dx, dy);
                 if (sp_KeyPressed(cambioBorde))
                     if (borde == 1)
                     {
                             borde = 2;
                             sp_Border(RED);
                     }
                     else
                     {
                             borde = 1;
                             sp_Border(BLUE);
                     }
           }
}
```

```
#asm

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._bicho1
defb @00000011, @11111100
defb @00000100, @11111000
defb @00001000, @11110000
defb @00001011, @11110000
defb @00001011, @11110000
defb @00001000, @11110000
defb @00001000, @11110000
defb @00000100, @11111000

defb @00000011, @11111100
defb @00001100, @11110011
defb @00001100, @11110011
defb @00011000, @11100111
defb @00011000, @11100111
defb @01111100, @10000011
defb @01111100, @10000011
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._bicho2
defb @11100000, @00011111
defb @00010000, @00001111
defb @00001000, @00000111
defb @01101000, @00000111
defb @01101000, @00000111
defb @00001000, @00000111
defb @10001000, @00000111
defb @10010000, @00001111

defb @11100000, @00011111
defb @00011000, @11100111
defb @00011000, @11100111
defb @00001100, @11110011
defb @00001100, @11110011
defb @00111110, @11000001
defb @00111110, @11000001
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
```

```
defb @00000000, @11111111
defb @00000000, @11111111

._bicho3
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

#endasm
```

Como veremos más adelante, deberemos implementar algún mecanismo para limitar la zona por donde nuestros sprites se van a desplazar. La forma de hacer esto es mediante simples comparaciones, comprobando que el lugar al que vamos a desplazar el sprite no este fuera de la pantalla (o de la zona donde queremos que permanezca).

AÑADIENDO COLOR

Un paso importante para poder tener como resultado un videojuego vistoso y que entre por los ojos es aprovechar los colores de nuestro Spectrum y disponer de una combinación agradable de cromaticidad en los sprites de nuestro juego. Los colores también permitirán distinguir con mayor facilidad nuestro personaje de los enemigos y el resto de elementos de la pantalla. Ya vimos en entregas anteriores que en el caso de los tiles era bastante sencillo modificar la tonalidad de la tinta y el papel:

```
sp_TileArray(' ', hash);
sp_Initialize(INK_WHITE | PAPER_BLACK, ' ');
```

pero en el caso de los sprites la cosa se complica un poco más y vamos a tener que dar unas cuantas "vueltas" para llegar a nuestro objetivo. En concreto, deberemos hacer uso de una función llamada `sp_IterateSprChar`, que recibe como parámetro una variable de tipo `struct sp_SS`, que representa el sprite que queremos modificar, y el nombre de una función que habremos definido anteriormente. Dicha función, al ser llamada, obtendrá como parámetro una variable de tipo `struct sp_CS`, una estructura muy interesante que nos va a permitir modificar diversas características del sprite, entre ellas el color.

El siguiente código muestra, en color rojo, las modificaciones realizadas al programa anterior para poder añadirle color al sprite de nuestro bicho, y que pasaremos a explicar a continuación:

```
#include <spritepack.h>
#include <stdlib.h>
#pragma output STACKPTR=61440
```

```
extern struct sp_Rect *sp_ClipStruct;
#asm
LIB SPCClipStruct
._sp_ClipStruct         defw SPCClipStruct
#endasm

extern uchar *sp_NullSprPtr;
#asm
LIB SPNullSprPtr
._sp_NullSprPtr         defw SPNullSprPtr
#endasm

extern uchar bicho1[];
extern uchar bicho2[];
extern uchar bicho3[];
struct sp_UDK keys;

uchar hash[] = {0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa};

void *my_malloc(uint bytes)
{
           return sp_BlockAlloc(0);
}

void *u_malloc = my_malloc;
void *u_free = sp_FreeBlock;


uchar n;
void addColour(struct sp_CS *cs)
{
           if (n == 0) // Se rellena de arriba a abajo y de izquierda
                                // a derecha, incluyendo partes vacias
del sprite
                cs->colour = INK_BLACK | PAPER_WHITE;
           else if (n == 1)
                cs->colour = INK_BLUE | PAPER_BLACK;
           else if (n == 2)
                 cs->colour = INK_RED | PAPER_GREEN;
           else if (n == 3)
                 cs->colour = INK_YELLOW | PAPER_BLACK;
           else if (n == 4)
                 cs->colour = INK_GREEN | PAPER_WHITE;
           else
                 cs->colour = TRANSPARENT;
           if (n > 5)
                 cs->graphic = sp_NullSprPtr;
           n++;
           return;
}


main()
{
           char dx,dy,i
           struct sp_SS *spriteBicho;
           uint reset,cambioBorde;
           int borde = 1;


            #asm
            di
            #endasm
```

```
              sp_InitIM2(0xf1f1);
              sp_CreateGenericISR(0xf1f1);
              #asm
              ei
              #endasm

             sp_TileArray(' ', hash);
             sp_Initialize(INK_WHITE | PAPER_BLACK, ' ');
             sp_Border(BLUE);
             sp_AddMemory(0, 255, 14, 0xb000);

             keys.up = sp_LookupKey('q');
             keys.down = sp_LookupKey('a');
             keys.right = sp_LookupKey('p');
             keys.left = sp_LookupKey('o');
             keys.fire = sp_LookupKey(' ');
             reset = sp_LookupKey('r');
             cambioBorde = sp_LookupKey('b');

             spriteBicho = sp_CreateSpr(sp_MASK_SPRITE, 3, bicho1, 1,
TRANSPARENT);
             sp_AddColSpr(spriteBicho, bicho2, TRANSPARENT);
             sp_AddColSpr(spriteBicho, bicho3, TRANSPARENT);
             sp_IterateSprChar(spriteBicho, addColour);
             sp_MoveSprAbs(spriteBicho, sp_ClipStruct, 0, 10, 15, 0, 0);

             while(1) {
                   sp_UpdateNow();


                   i  = sp_JoyKeyboard(&keys);
                   if ((i & sp_FIRE) == 0) {
                        dx = dy = 1;
                   } else {
                        dx = dy = 8;
                   }
                   if ((i & sp_LEFT) == 0)
                        dx = -dx;
                   else if ((i & sp_RIGHT) != 0)
                        dx = 0;
                   if ((i & sp_UP) == 0)
                        dy = -dy;
                   else if ((i & sp_DOWN) != 0)
                        dy = 0;
                   if (sp_KeyPressed(reset))
                       sp_MoveSprAbs(spriteBicho, sp_ClipStruct, 0, 10, 15,
0, 0);
                   else
                       sp_MoveSprRel(spriteBicho, sp_ClipStruct, 0, 0, 0,
dx, dy);
                   if (sp_KeyPressed(cambioBorde))
                       if (borde == 1)
                       {
                               borde = 2;
                               sp_Border(RED);
                       }
                       else
                       {
                               borde = 1;
                               sp_Border(BLUE);
                       }
             }
}
```

```
#asm

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._bicho1
defb @00000011, @11111100
defb @00000100, @11111000
defb @00001000, @11110000
defb @00001011, @11110000
defb @00001011, @11110000
defb @00001000, @11110000
defb @00001000, @11110000
defb @00000100, @11111000

defb @00000011, @11111100
defb @00001100, @11110011
defb @00001100, @11110011
defb @00011000, @11100111
defb @00011000, @11100111
defb @01111100, @10000011
defb @01111100, @10000011
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._bicho2
defb @11100000, @00011111
defb @00010000, @00001111
defb @00001000, @00000111
defb @01101000, @00000111
defb @01101000, @00000111
defb @00001000, @00000111
defb @10001000, @00000111
defb @10010000, @00001111

defb @11100000, @00011111
defb @00011000, @11100111
defb @00011000, @11100111
defb @00001100, @11110011
defb @00001100, @11110011
defb @00111110, @11000001
defb @00111110, @11000001
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
```

```
._bicho3
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

#endasm
```

Como se puede comprobar, tras crear el sprite del bicho, hacemos uso de la siguiente instrucción:

```
sp_IterateSprChar(spriteBicho, addColour);
```

Esto significa que vamos a usar una función llamada `addColour`, y que deberemos haber definido anteriormente dentro del archivo con el código, para modificar las propiedades de `spriteBicho`. Aunque la instrucción solo aparece una vez, es como si estuviéramos llamando a la función `addColour` una vez por cada bloque de 8x8 que forma el sprite del bicho (recordemos que el sprite de nuestro bicho está formado por 3 columnas de 3 bloques de 8x8, por lo que la función `addColour` será llamada un total de 9 veces). La forma que tiene está función es la siguiente:

```
uchar n;
void addColour(struct sp_CS *cs)
{
           if (n == 0) // Se rellena de arriba a abajo y de izquierda
                                 // a derecha, incluyendo partes vacias
del sprite
                cs->colour = INK_BLACK | PAPER_WHITE;
           else if (n == 1)
                cs->colour = INK_BLUE | PAPER_BLACK;
           else if (n == 2)
                 cs->colour = INK_RED | PAPER_GREEN;
           else if (n == 3)
                 cs->colour = INK_YELLOW | PAPER_BLACK;
           else if (n == 4)
                 cs->colour = INK_GREEN | PAPER_WHITE;
           else
                 cs->colour = TRANSPARENT;
           if (n > 5)
```

```
                        cs->graphic = sp_NullSprPtr;
                n++;
                return;
    }
```

Justo antes se define una variable global llamada n, y que no será más que un contador que nos permitirá saber en cuál de las nueve llamadas a la función addColour nos encontramos. Ya dentro de dicho método se observa como su valor se incrementa de uno en uno en cada llamada.

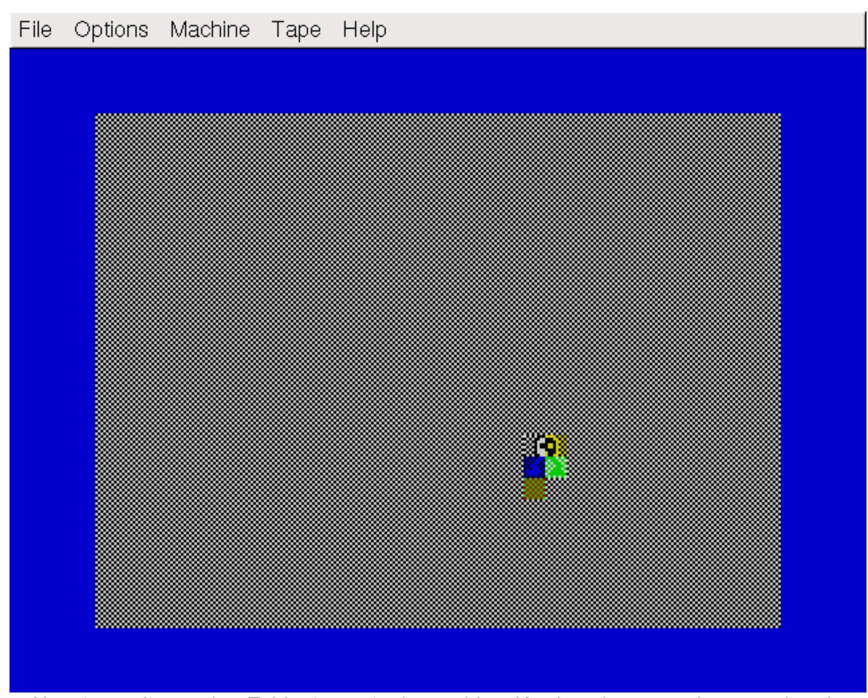
Como hemos repetido varias veces, la función addColour será llamada una vez por cada bloque 8x8 que forme nuestro sprite, recibiendo como parámetro un struct de tipo sp_CS que nos va a permitir modificar diversas características de dicho bloque del sprite. Uno de los campos de ese struct es colour, que como su propio nombre indica, es el indicado para añadir color. Gracias al valor de n, podremos conocer en cuál de todos los bloques del sprite nos encontramos (empezando por el 0, los bloques están ordenados de arriba a abajo y de izquierda a derecha, por lo que en el caso de nuestro bicho, aun estando compuesto por 3x3 bloques, solo nos interesará colorear aquellos para los que n vale 0,1,3 y 4, que son los que no están vacíos) y asignarle un color de tinta y papel modificando el valor del campo colour del struct sp_CS, tal como se puede observar en el código anterior.

Solo deberemos colorear los bloques 0,1,3 y 4 de nuestro bicho, pues el bloque 2 se corresponde con el último de la primera columna (que está vacío), el bloque 5 con el último de la segunda columna (que también está vacío) y los bloques 6,7 y 8 con la última columna de todas, también vacía, y que se añadió para que no hubiera problemas al desplazar el sprite. En el caso de los bloques 2 y 5 lo más correcto hubiera sido utilizar el valor TRANSPARENT para el campo colour (aunque en nuestro ejemplo hemos sido un poco transgresores y el bloque 2 no lo hemos hecho transparente). Para la última columna (bloques para los que n vale más que 5), sin embargo, asignamos el valor sp_NullSprPtr al campo colour. Este valor, que ha sido definido anteriormente en el programa de la siguiente forma:

```
extern uchar *sp_NullSprPtr;
#asm
LIB SPNullSprPtr
._sp_NullSprPtr          defw SPNullSprPtr
#endasm
```

evitará que esa columna vacía "moleste" al resto de los sprites con los que nuestro bicho entre en contacto.

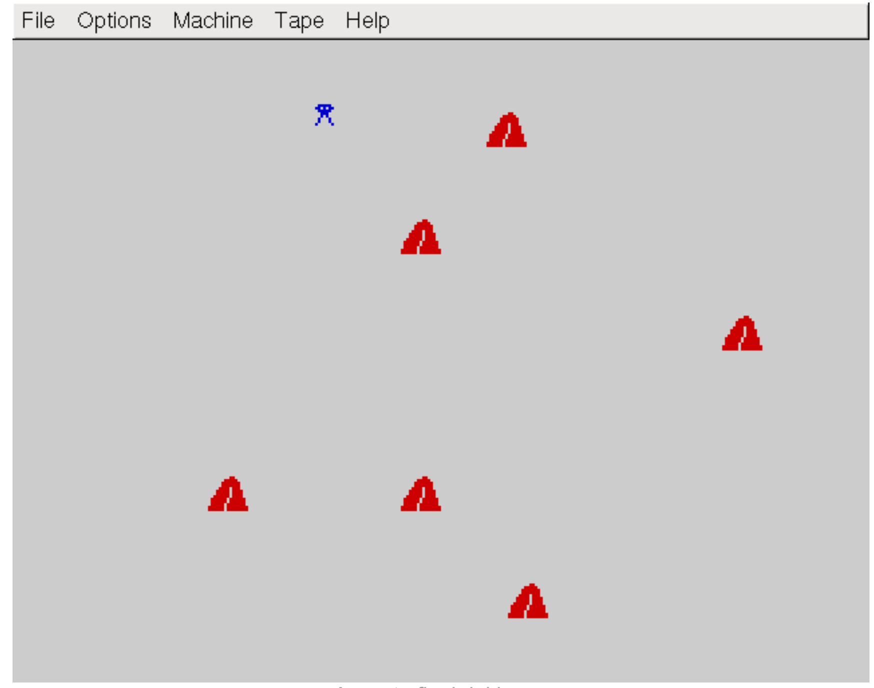
Y ya está, ya le hemos dado color a nuestro bicho (eso sí, una combinación bastante psicodélica). Cada vez que queramos hacer lo mismo con cualquier otro sprite, tan solo deberemos seguir la receta anterior, pues es algo mecánico.



Nuestro sprite a color. Evidentemente, la combinación de colores puede ser mejorada

UN NUEVO JUEGO

A continuación, y empleando el conocimiento adquirido tanto en este número como en el número anterior del magazine, vamos a comenzar a desarrollar nuestro propio juego. Al hacerlo nos encontraremos con una serie de problemas que iremos resolviendo en próximos artículos. En nuestro juego controlaremos los movimientos de un esquiador que deberá descender una montaña esquivando todas las rocas que se interpongan en su camino. El esquiador estará situado en la parte superior de la pantalla y podrá desplazarse a izquierda y derecha. Las rocas subirán desde la parte inferior de la pantalla, simulando el descenso por la pendiente nevada de la montaña.



Aspecto final del juego

El código completo, que comentaremos a lo largo de esta sección (aunque no en detalle, sólo aquellas partes que difieran de lo visto hasta ahora o introduzcan conceptos nuevos), es el siguiente:

```
#include <spritepack.h>
#include <stdlib.h>
#pragma output STACKPTR=61440

#define NUM_ROCAS 8

extern struct sp_Rect *sp_ClipStruct;
#asm
LIB SPCClipStruct
._sp_ClipStruct         defw SPCClipStruct
#endasm

extern uchar *sp_NullSprPtr;
#asm
LIB SPNullSprPtr
._sp_NullSprPtr         defw SPNullSprPtr
#endasm

extern uchar roca1[];
extern uchar roca2[];
extern uchar roca3[];
extern uchar banderin1[];
extern uchar banderin2[];
extern uchar skiCentrado1[];
extern uchar skiCentrado2[];
extern uchar skiIzquierda1[];
extern uchar skiIzquierda2[];
extern uchar skiDerecha1[];
```

```
extern uchar skiDerecha2[];
struct sp_UDK keys;

void *my_malloc(uint bytes)
{
            return sp_BlockAlloc(0);
}

void *u_malloc = my_malloc;
void *u_free = sp_FreeBlock;

uchar n;
void addColourRoca(struct sp_CS *cs)
{
            if (n >= 0 && n <= 5)
                  cs->colour = INK_RED | PAPER_WHITE;
            else
                  cs->colour = TRANSPARENT;
            if (n > 5)
                  cs->graphic = sp_NullSprPtr;
            n++;
            return;
}

void addColourSki(struct sp_CS *cs)
{
        if (n == 0)
                cs->colour = INK_BLUE | PAPER_WHITE;
        else if (n == 2)
                cs->colour = INK_BLUE | PAPER_WHITE;
        else
                cs->colour = TRANSPARENT;
        if (n>2)
                cs->graphic = sp_NullSprPtr;
        n++;
        return;
}

main()
{
            char dx,dy,i
            struct sp_SS *spriteRoca[NUM_ROCAS], *spriteSkiCentrado,
                 *spriteSkiIzquierda, *spriteSkiDerecha, *ski;
            short int posicion = 0;
            int roca = 0;


             #asm
             di
             #endasm
             sp_InitIM2(0xf1f1);
             sp_CreateGenericISR(0xf1f1);
             #asm
             ei
             #endasm

            sp_Initialize(INK_BLACK | PAPER_WHITE, ' ');
            sp_Border(WHITE);
            sp_AddMemory(0, 255, 14, 0xb000);

            keys.up = sp_LookupKey('q');
            keys.down = sp_LookupKey('a');
            keys.fire = sp_LookupKey(' ');
            keys.right = sp_LookupKey('p');
            keys.left = sp_LookupKey('o');
```

```
          for (i=0;icol > 0)
                {
                   if (posicion != -1)
                   {
                       sp_MoveSprAbs(spriteSkiIzquierda,sp_ClipStruct,0,
ski->row,ski->col,0,0);
                       sp_MoveSprAbs(ski,sp_ClipStruct,0,0,-10,0,0);
                       ski = spriteSkiIzquierda;
                       posicion = -1;
                   }
                   dx = -3;
                }
                else if ((i & sp_RIGHT) == 0 && ski->col < 30)
                {
                   if (posicion != 1)
                   {
                       sp_MoveSprAbs(spriteSkiDerecha,sp_ClipStruct,0,sk
i->row,ski->col,0,0);
                       sp_MoveSprAbs(ski,sp_ClipStruct,0,0,-10,0,0);
                       ski = spriteSkiDerecha;
                       posicion = 1;
                   }
                   dx = 3;
                }
                else
                {
                   if (posicion != 0)
                   {
                           sp_MoveSprAbs(spriteSkiCentrado,sp_ClipStruct
,0,ski->row,ski->col,0,0);
                           sp_MoveSprAbs(ski,sp_ClipStruct,0,0,-10,0,0);
                           ski = spriteSkiCentrado;
                           posicion = 0;
                   }
                }
                if (dx != 0) sp_MoveSprRel(ski, sp_ClipStruct, 0, 0, 0,
dx, dy);

                if (spriteRoca[roca]->row != -10)
                {
                   dx = 0;
                   dy = -4;
                   sp_MoveSprRel(spriteRoca[roca],sp_ClipStruct,0,0,0,dx
,dy);
                }
                else
                   if (rand()%100>98)
                   {
                       sp_MoveSprAbs(spriteRoca[roca],sp_ClipStruct,0,23
,rand()%29+1,0,4);
                   }
                roca ++;
                if (roca >= NUM_ROCAS)
                   roca = 0;
          }
}

#asm

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
```

```
defb @00000000, @11111111
defb @00000000, @11111111

._roca1
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000001, @11111110
defb @00000011, @11111100
defb @00000011, @11111100
defb @00000111, @11111000

defb @00001111, @11110000
defb @00001111, @11110000
defb @00011111, @11100000
defb @00111111, @11000000
defb @00111111, @11000000
defb @00111110, @11000000
defb @01111110, @10000000
defb @01111110, @10000000

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._roca2
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @01100000, @10011111
defb @11110000, @00001111
defb @11111000, @00000111
defb @11111000, @00000111
defb @10111000, @00000111

defb @10111100, @00000011
defb @10111100, @00000011
defb @10111100, @00000011
defb @01111110, @00000001
defb @01111110, @00000001
defb @11111110, @00000001
defb @11111111, @00000000
defb @11111111, @00000000

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._roca3
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
```

```
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._skiCentrado1
defb @00111110, @11000001
defb @01101011, @10000000
defb @00111110, @11000001
defb @00011100, @11100011
defb @00010100, @11101011
defb @00100010, @11011101
defb @00100010, @11011101
defb @01000001, @10111110

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._skiCentrado2
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._skiIzquierda1
defb @00011111, @11100000
defb @00110101, @11000000
defb @00011111, @11100000
defb @00001110, @11110001
```

```
defb @00010010, @11101101
defb @00100100, @11011011
defb @01001000, @10110111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._skiIzquierda2
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._skiDerecha1
defb @11111000, @00000111
defb @10101100, @00000011
defb @11111000, @00000111
defb @01110000, @10001111
defb @01001000, @10110111
defb @00100100, @11011011
defb @00010010, @11101101
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

._skiDerecha2
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
```

```
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111
defb @00000000, @11111111

#endasm
```

Se han definido un total de cuatro sprites. El sprite de tipo `roca` (dividido en tres columnas denominadas `roca1`, `roca2` y `roca3`) será el que representará los obstáculos que encontrará nuestro intrépido esquiador en su descenso. Para el protagonista de nuestro juego se han definido tres sprites divididos cada uno de ellos en dos columnas: `skiCentrado` que será el que se mostrará cuando el esquiador esté descendiendo sin que lo movamos a izquierda o derecha, `skiIzquierda` que representa nuestro esquiador desplazándose hacia la izquierda, y `skiDerecha` que a su vez representa al mismo esquiador desplazándose a la derecha.

Como debemos reservar memoria para los sprites que vayamos a mostrar por pantalla, lo haremos para cada una de las tres posiciones del esquiador (de tal forma que mostraremos la adecuada según el movimiento del jugador) y un número fijo de rocas. Es probable que no todas las rocas sean visibles simultáneamente durante el juego, pero hacerlo de esta forma simplifica en gran medida el diseño, sin tener que hacer grandes esfuerzos para reservar o liberar memoria cuando sea necesario. En concreto, la constante NUM_ROCAS es la que vamos a usar para indicar el número de rocas que se van a crear. Se define un array llamado `spriteRoca` con tantas posiciones como las indicadas por la constante anteriormente mencionada, y en cada posición de dicho array se reserva memoria para un sprite de tipo roca que es coloreado con su correspondiente función `addColour`:

```
for (i=0;i
```

Un par de comentarios sobre la definición de los sprites en este programa: como primer parámetro para los sprites de tipo roca en la función `sp_CreateSpr` se pasa el valor `sp_OR_SPRITE` en lugar de `sp_MASK_SPRITE`. Esto permite mayor velocidad en el dibujado de las rocas por la pantalla a cambio de renunciar a que el sprite pueda tener píxeles transparentes en su interior. Por otra parte, antes de llamar a la correspondiente función addColour para cada sprite se vuelve a dar a la variable global `n` el valor 0, de tal forma que se pueda iterar de nuevo por todos los bloques de cada nuevo sprite que queramos colorear. Esto se puede ver también en el caso de los tres sprites para el esquiador.

Una vez creados los sprites se utilizan las siguientes instrucciones:

```
        i = spriteSkiCentrado;

        sp_MoveSprAbs(ski, sp_ClipStruct, 0, 0, 15, 0, 0);
```

El puntero `ski` apuntará en todo momento al sprite del esquiador que deba ser mostrado por pantalla. El esquiador comenzará mirando hacia abajo, ya que no se está moviendo ni a izquierda ni a derecha; por lo tanto, `ski` apuntará al sprite `spriteSkiCentrado`, que es el que deberá ser visualizado. Cada vez que el esquiador se mueva a izquierda o derecha, `ski` apuntará, respectivamente, a `spriteSkiIzquierda` y `spriteSkiDerecha`.

En el bucle principal iniciado por la sentencia `while(1)` y que se ejecutará de forma infinita, vemos como se resuelve el tema de la animación del protagonista, la limitación de sus movimientos por la pantalla, y el movimiento de las rocas. Comencemos por la primera parte de dicho bucle, que es donde encontramos el código referido al movimiento del esquiador:

```
                  i = sp_JoyKeyboard(&keys);
                dx = 0;
                dy = 0;
                if ((i & sp_LEFT) == 0 && ski->col > 0)
                {
                   if (posicion != -1)
                   {
                       sp_MoveSprAbs(spriteSkiIzquierda,sp_ClipStruct,0,ski-
>row,ski->col,0,0);
                       sp_MoveSprAbs(ski,sp_ClipStruct,0,0,-10,0,0);
```

```
                    ski = spriteSkiIzquierda;
                  posicion = -1;
               }
               dx = -3;
            }
            else if ((i & sp_RIGHT) == 0 && ski->col < 30)
            {
               if (posicion != 1)
               {
                  sp_MoveSprAbs(spriteSkiDerecha,sp_ClipStruct,0,ski-
>row,ski->col,0,0);
                  sp_MoveSprAbs(ski,sp_ClipStruct,0,0,-10,0,0);
                  ski = spriteSkiDerecha;
                  posicion = 1;
               }
               dx = 3;
            }
            else
            {
               if (posicion != 0)
               {
                     sp_MoveSprAbs(spriteSkiCentrado,sp_ClipStruct,0,ski-
>row,ski->col,0,0);
                     sp_MoveSprAbs(ski,sp_ClipStruct,0,0,-10,0,0);
                     ski = spriteSkiCentrado;
                     posicion = 0;
               }
            }
            if (dx != 0) sp_MoveSprRel(ski, sp_ClipStruct, 0, 0, 0, dx,
dy);
```

Cada vez que el esquiador cambia de posición, se debe hacer apuntar a ski hacia el sprite adecuado. Esto solo se debe hacer en la primera iteración en la que estemos moviéndonos en esa dirección. Para ello se observa que se utiliza una variable posicion, inicializada a cero, y que valdrá precisamente 0 si el sprite a mostrar por pantalla debe ser spriteSkiCentrado, y -1 o 1 si se debe mostrar el esquiador desplazándose hacia la izquierda o la derecha, respectivamente. Este valor nos va a permitir saber cuándo debemos hacer que el puntero ski apunte hacia otro sprite diferente del que lo estaba haciendo hasta ese momento.

Veámoslo con un ejemplo. Supongamos que el esquiador se está deslizando ladera abajo, sin que el jugador lo mueva a izquierda o derecha. En este caso, ski apunta a spriteSkiCentrado, y posicion vale 0. A continuación, el jugador pulsa la tecla de la derecha y la mantiene presionada. En la primera iteración en la que esto sucede, se realizan las siguientes acciones:

- Se mueve spriteSkiDerecha, que representa al esquiador desplazándose en esa dirección, a la misma posición de la pantalla donde se encuentra el sprite del esquiador centrado, que es el que se estaba mostrando hasta ahora. Para conocer esa posición se hace uso de los campos row y col de la estructura ski.
- A continuación, el sprite apuntado por ski se mueve fuera de la pantalla, pues no va a ser mostrado más.
- La variable ski apunta a spriteSkiDerecha, para poder mostrarlo por la pantalla.
- Se cambia el valor de posicion a 1, indicando que ya hemos estado al menos una iteración desplazándonos hacia la derecha y que no es necesario repetir todas estas operaciones en el caso en el que sigamos moviéndonos en esta dirección.

Sólo podremos desplazarnos hacia la izquierda o la derecha siempre que nos encontremos dentro de los límites de la pantalla. Esto se controla de la siguiente forma:

```
if ((i & sp_LEFT) == 0 && ski->col > 0)

else if ((i & sp_RIGHT) == 0 && ski->col < 30)
```

Con respecto a las rocas, moveremos tan solo una en cada iteración del bucle principal. Esto no es más que una medida preliminar para conseguir un mínimo de velocidad y algo de sincronización, pero

es evidente que deberemos modificar este apartado del código más adelante. Para poder desplazar tan solo una roca en cada iteración, hacemos uso de la variable `roca`, inicializada a cero, que nos va a indicar cual de todas las rocas va a ser movida. Esta variable se incrementa en 1 cada vez, volviéndole a asignar el valor 0 cuando almacena un entero superior al valor indicado por NUM_ROCAS. Finalmente, y como se puede observar en el código siguiente, esta variable `roca` se utiliza como indice del array de sprites de tipo roca, determinando qué roca se mueve.

```
                    if (spriteRoca[roca]->row != -10)
                    {
                       dx = 0;
                       dy = -4;
                       sp_MoveSprRel(spriteRoca[roca],sp_ClipStruct,0,0,0,dx
,dy);
                    }
                    else
                       if (rand()%100>98)
                       {
                           sp_MoveSprAbs(spriteRoca[roca],sp_ClipStruct,0,23
,rand()%29+1,0,4);
                       }
                    roca ++;
                    if (roca >= NUM_ROCAS)
                       roca = 0;
```

Cada roca se desplazará hacia arriba en la pantalla usando un valor de desplazamiento `dy = -4`, hasta llegar a un valor de -10 en su coordenada y. En ese momento, la roca ya habrá desaparecido de la pantalla por su parte superior. Si una roca está fuera de la pantalla, volveremos a sacarla por debajo de nuevo en una posición x al azar (como si fuera otra roca distinta) con una probabilidad del 98% (para que una roca que ha salido por la parte superior de la pantalla no vuelva a aparecer inmediatamente por la parte inferior y que efectivamente parezca una roca diferente). Se podría modificar el valor de la constante `NUM_ROCAS` para cambiar el número de obstáculos, pero se ha escogido un valor que hace que el juego no vaya demasiado lento.

Una vez que compilamos y ejecutamos nuestro juego, nos encontramos con tres problemas, que trataremos de resolver en artículos sucesivos:

- No hay colisiones. Nuestro esquiador parece un fantasma, pues puede atravesar las rocas sin que éstas le afecten. Eso es porque no hemos establecido un control de colisiones.
- No hay sincronización. Esto quiere decir que la velocidad del juego es diferente según el número de rocas que se estén mostrando en determinado momento por la pantalla. Haciendo uso de las interrupciones podremos conseguir que haya el número de rocas que haya, la velocidad de descenso se mantenga constante.
- Colour clash. Cuando nuestro esquiador y las rocas entran en contacto, se produce un choque de atributos bastante desagradable (los colores de los sprites parecen "mezclarse"). Esto también se puede medio resolver.

Por último, un apunte sobre la liberación de memoria. Como hemos visto, cada vez que queremos mostrar un sprite por pantalla, debemos reservar memoria para el mismo. En el caso en el que un sprite no vaya a ser mostrado nunca más, lo más adecuado es liberar la memoria correspondiente, tras haber movido dicho sprite fuera de la pantalla para evitar efectos no deseados.

Y EN EL PRÓXIMO NUMERO

¡La programación de nuestro juego ha comenzado! Sin embargo, nos encontramos con varios problemas, tal como se ha comentado anteriormente; problemas que intentaremos resolver en sucesivas entregas del curso centrándonos en tres aspectos: cómo solucionar el problema conocido como colour clash, el uso de IM2 (interrupciones), y la intersección de rectángulos para detección de colisiones. Gracias a estos elementos nuestro juego quedará mucho más completo y jugable. Con unos pocos añadidos más podría incluso compararse a las grandes producciones comerciales de 1983 ;).

LINKS

Código fuente

SIEW

ENTREVISTA A ALFONSO FERNÁNDEZ BORRO «BORROCOP»

En esta ocasión entrevistamos a uno de los más prolíficos y grandes grafistas que nuestro Spectrum ha podido tener. Suyos son los gráficos de juegos como Viaje al Centro de la Tierra, Gremlins 2 o Zona 0, entre otros. Y próximamente nos sorprenderá de nuevo con una revisión para PC de un trabajo propio: un remake de R.A.M.

¿Cuándo pusiste tus manos en un ordenador por vez primera?

La primera vez que puse mis manos fue en casa de mi cuñado que se había comprado un Spectrum para llevar una base de datos y de regalo venían unos juegos en "casete" que eran casi igualitos a los juegos de las máquinas de videojuegos de los salones recreativos, fue mi primera experiencia allá por el año 1984, por lo que ya ha llovido lo suyo...

Y tu faceta de dibujante, ¿cómo surgió y cuándo la trasladaste a un ordenador?

Pues surgió desde que tenía uso de razón ya dibujaba en las paredes de mi casa, cosa que no debió ser de buen gusto para mis padres, que aprovecharon la ocasión para ilustrarme bien el trasero con la zapatilla de mi madre, he de decir que por lo que podéis comprobar no consiguieron desviar mi pasión por el dibujo.

Toda esta pasión me llevó dirigir mi carrera hacia el dibujo, quería haber sido arquitecto pero al acabar C.O.U. mi padre me dijo que la situación familiar no daba para pagarme la carrera y tuve que bajar un escalón empezando a estudiar delineación. Cuando estaba en el último curso, poco antes de irme a la mili, vi que había unas máquinas llamadas "plotters" que se cepillaban un plano de dibujo en menos de cinco minutos, mientras que yo ese mismo plano hubiera tardado alrededor de una semana. Entonces pensé: "debo dejar lo que estoy haciendo y ponerme a estudiar como funcionan estos infernales aparatos".

Se te conoce esencialmente por tu faceta de dibujante y grafista. Adicionalmente, ¿participabas de alguna manera en cada proyecto de otro modo: ya sea programando, diseñando...?

Si, hacía muchas más cosas, diseñaba niveles, mapeados, en muchos casos también en la elaboración de los proyectos dando ideas a la vez que al finalizar hacia de betatester de los juegos de mis compañeros.

Durante tu trayectoria profesional como diseñador gráfico has podido trabajar con diferentes equipos de trabajo. ¿Variaba tu forma de trabajar con cada uno de ellos?

> "Siempre trabajé y trabajo con la máxima de tener libertad a la hora de desarrollar mis ideas."

Hombre, pues claro está que sí, aunque digamos que siempre trabajé y trabajo con la máxima de tener libertad a la hora de desarrollar mis ideas, aunque a posteriori luego haya los cambios oportunos.

¿Tenías completa libertad creativa para realizar los gráficos de los diferentes juegos que te encomendaban?

Normalmente sí, aunque casos como Gremlins 2 te obligaban a tener que basarte en unos modelos que ya estaban creados ya que eran protagonistas de la propia película.

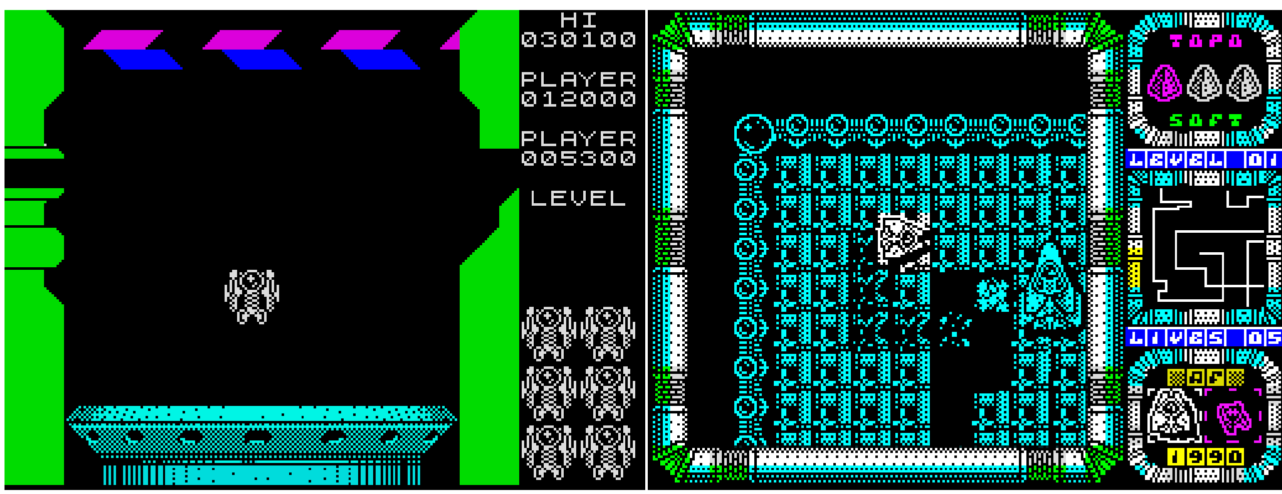
Es ciertamente frustrante haber hecho unos gráficos alucinantes y que éstos sean rechazados por imposibilidad de implementarlos o por no gustarle al jefe. ¿Te ocurrió alguna vez esto con algunos de tus diseños?

No exactamente más bien se rechazaban del proyecto entero porque se cambiaba la filosofía del juego y la perspectiva de este, como ocurrió con la

continuación de R.A.M., que, después de tener el proyecto muy avanzado gráficamente, se cambió la perspectiva de cenital a isométrica afectando incluso al nombre del juego, pasando de ser la continuación de R.A.M. a Tronner; y por no poder utilizar ese nombre al no llegar a un acuerdo con Disney, a llamarse Zona 0 que es como todos lo conocisteis tiempo después.

Aparte del diseño sobre el papel, ¿también hacías las pertinentes conversiones finales en los ordenadores (Spectrum, Amstrad...)? ¿Qué programas usabas para ello?

Bueno digamos aquí que utilizábamos primero un soporte intermedio para las conversiones que era el Atari ST. Una vez que teníamos los gráficos finales en Spectrum pasábamos estos al Atari ST mediante conexión RS232-paralelo y en éste utilizábamos los programas de diseño Keating y Degas (creo que se llamaban así, ya no lo recuerdo bien) en donde pasábamos el típico píxel cuadrado de Spectrum al ladrillazo del CPC empezando también a darle color. Cuando lo teníamos mas o menos desarrollado lo volvíamos a pasar, esta vez del Atari ST al Amstrad CPC donde le dábamos los retoques finales con los programas Screen Designer u otro que teníamos parcheado por allí.



Atomito fotón (Izda.) y R.A.M. 2, dos juegos que no llegaron a ver la luz

¿Cómo era un día normal de trabajo en Topo Soft: teníais un horario flexible, ambiente distendido... o todo lo contrario?

El día normal era tan normal como el de cualquier trabajador, el horario no era nada flexible sino que teníamos que firmar tanto a la salida como la entrada de la jornada de 8 horas. En cuanto al ambiente, era distendido en un principio, pero según pasaron los años, fue haciéndose cada vez menos respirable, hasta el final en el que aquello parecía más un ministerio y nosotros meros funcionarios.

La entrada de Gabriel Nieto en Topo Soft marcó un punto de inflexión en la compañía, pasando a convertirse en una auténtica empresa. ¿Cambió esto vuestra forma de trabajar en los proyectos? ¿Se os exigía más?

Esto es como el fútbol que cuando se cambia de entrenador se cambia la forma de trabajar. Quizás Gabriel Nieto fuera menos entendido en como se hacían los juegos desde abajo, como lo era Javier Cano que lo venía mamando desde un principio, pero al contrario que éste, en lo comercial y el diseño era un auténtico formula 1, por la que Gabriel consiguió un equilibrio entre lo que se hacía y lo que se vendía, cosa que Javier no fue nunca capaz. Los juegos de Javier fueron casi siempre mejores, más jugables incluso, pero Gabriel fue capaz de venderles polos a los esquimales. Como caso puede decirse el "Score 3020" un pinball injugable pero que gracias a él y a la manera de llevar el marketing tuvo unas excelentes ventas para la mierda que era.

Varios de vuestros programas, como, por ejemplo, Viaje al Centro de la Tierra, fueron muy bien recibidos tanto por el público como por la critica especializada. ¿Cómo recibíais este tipo de halagos dentro del equipo de desarrollo?

La verdad es que en esta caso fue un tema un poco agridulce. Acababan de despedir a la mitad de la plantilla y de los integrantes del proyecto, con lo que fue un halago empañado por la tristeza del que no pudimos reponernos en mucho tiempo. En cuanto a otros títulos, pues nos llenaba de satisfacción comprobar que el esfuerzo había merecido la pena, viendo que la gente les gustaba tanto como a nosotros hacerlo.

¿Y el caso contrario: recuerdas si hubo algún juego que pensarais que iba a ser un bombazo y no ocurrió así?

No, se sabía de antemano que programas iban a tener tirón y cuales no pasarían de ser verdaderas bazofias dignas de no haber salido nunca al

mercado, pero claro está que había que recuperar la inversión.

Evidentemente lo primero en que te fijarás de un juego, es en sus gráficos. ¿había algún diseñador de 8 bits del que tuvieras "sana" envidia?

Sana sana pero sanísima envidia de mi buen amigo Jorge Azpiri el mejor de aquella época y el mejor también después en 16 bits. Además viéndole trabajar aprendías un montón y era compañerísimo ayudándote en todo momento y aportaba ideas para que los demás hiciéramos un trabajo con más calidad.

Actualmente, además de tu trabajo profesional, te has volcado en el desarrollo de 'remakes', trabajando en un juego tuyo ("RAM 2") haciendo los gráficos. ¿Cómo surgió esta posibilidad?

Bueno, encontré un grupo de guerreros que no quieren tirar sus viejas máquinas y que viven en un pequeño país llamado CEZ GS, dentro del mundo de Computer Emuzone, a los que me he unido para sacar del olvido algunos proyectos que no llegaron a salir al público como ese RAM 2 del que antes os hablé.

¿En qué otros proyectos estás involucrado ahora mismo?

Jejeje, en este mundillo no se pueden decir las cosas muy alto cuando todo está en pañales, pero puedo adelantaros que mis gráficos verán la luz antes de final de año para Pc… y hasta aquí puedo leer, como decían en el 1, 2, 3...



Un adelanto gráfico del remake de R.A.M. para PC

¿Hay prevista alguna colaboración tuya en alguna versión de ordenadores de 8 bits?

En principio solamente estoy con lo de RAM 2, aunque muy por encima, ya que no tengo tiempo para hacer muchas cosas para estas máquinas. Aunque pienso que algo haré cuando termine con lo de RAM 2.

Es curioso comprobar como hay gente que rechaza hablar, al contrario que tú, de su pasado como desarrollador de 8 bits. ¿Tan "olvidable" crees que fue aquella época?

Hay gente que pasó por este mundo del 8 bit como mero "mercenario del soft" y como eso no fue su guerra, no tiene nada que decir. Yo, por el contrario, disfruté de esa época y no tengo que esconderme, sino al contrario, disfrutar con vosotros compartiendo experiencias que es lo que hace que seamos entre todos más grandes y que este mundillo sea tan genial.

Tu currículum profesional es impresionante: Iber Soft, Topo Soft, Lucas Arts, Pyro Studios... ¿En cuál te sentiste más a gusto profesionalmente?

En Topo, claro está, fue el lugar en que desarrollé mis trabajos con más libertad, con un ambiente mucho más sano, entrañable y en donde conocí a muchos y muy buenos amigos.

¿Alguna compañía en la que te hubiera gustado trabajar, ya sea nacional o extranjera?

Por las nacionales pasé por casi todas. Quizás me hubiera gustado trabajar para Zigurat de las de aquí. En cuanto a las extranjeras no puedo traicionar a mis ideas, y, sin dudarlo, Ultimate, aunque tampoco me hubiera importado hacerlo para Vortex o Mikro-gen.

Gracias por contestar a nuestra entrevista y si quieres comentar algo más, éste es tu espacio...

Gracias a vosotros por dedicarme un tiempo de vuestra vida cuando jugabais con los juegos que hice, y sobre todo, gracias por esta entrevista que espero que os guste y la disfrutéis como yo la he disfrutado contestando lo más sinceramente posible. Seguro que estaremos en contacto.

HORACE

programación ensamblador

LENGUAJE ENSAMBLADOR DEL Z80 (I)

LA ARQUITECTURA DEL SPECTRUM

En esta tercera entrega del curso explicaremos la sintaxis utilizada en los programas en ensamblador. Para ello comenzaremos con una definición general de la sintaxis del lenguaje para el ensamblador Pasmo, que será el "traductor" que usaremos entre el lenguaje ensamblador y el código máquina del Z80.

Posteriormente veremos en detalle los registros: qué registros hay disponibles, cómo se agrupan, y el registro especial de Flags, enlazando el uso de estos registros con las instrucciones de carga, de operaciones aritméticas, y de manejo de bits, que serán las que trataremos hoy.

Esta entrega del curso es delicada y complicada: por un lado, tenemos que explicar las normas y sintaxis del ensamblador cruzado PASMO antes de que conozcamos la sintaxis del lenguaje ensamblador en sí, y por el otro, no podremos utilizar PASMO hasta que conozcamos la sintaxis del lenguaje.

Además, el lenguaje ensamblador tiene disponibles muchas instrucciones diferentes, y nos resultaría imposible explicarlas todas en un mismo capítulo, lo que nos fuerza a explicar las instrucciones del microprocesador en varias entregas. Esto implica que hablaremos de PASMO comentando reglas, opciones de instrucciones y directivas que todavía no conocemos.

Es por esto que recomendamos al lector que, tras releer los capítulos 1 y 2 de nuestro curso, se tome esta entrega de una manera especial, leyéndola 2 veces. La "segunda pasada" sobre el texto permitirá enlazar todos los conocimientos dispersos en el mismo, y que no pueden explicarse de una manera lineal porque están totalmente interrelacionados. Además, la parte relativa a la sintaxis de PASMO será una referencia obligada para posteriores capítulos (mientras continuemos viendo diferentes instrucciones ASM y ejemplos).

Así pues, os recomendamos leer este capítulo dos veces, una para absorber las normas de PASMO, y otra para absorber la sintaxis del lenguaje y las instrucciones que explicaremos (terminando de comprender conceptos de la primera lectura).

SINTAXIS DEL LENGUAJE EN PASMO

En la primera parte de curso se introdujo PASMO, el ensamblador cruzado que recomendamos para el desarrollo de programas para Spectrum. Este ensamblador traduce nuestros ficheros de texto .asm con el código fuente de programa (en lenguaje ensamblador) a ficheros .bin (o .tap/.tzx) que contendrán el código máquina directamente ejecutable por el Spectrum. Suponemos que ya tenéis instalado PASMO (ya sea la versión Windows o la de Linux) en vuestro sistema y que sabéis utilizarlo (os recomiendo que releáis la primera entrega de nuestro curso si no recordáis su uso), y que podéis ejecutarlo dentro del directorio de trabajo que habéis elegido (por ejemplo: /home/usuario/z80asm o C:\z80asm).

El ciclo de desarrollo con PASMO será el siguiente:

- Con un editor de texto, tecleamos nuestro programa en un fichero .ASM con la sintaxis que veremos a continuación.
- Salimos del editor de texto y ensamblamos el programa con: "pasmo ejemplo1.asm ejemplo1.bin".
- El fichero bin contiene el código máquina resultante, que podemos POKEar en memoria, cargar como un bloque CM (LOAD "" CODE) tras nuestro programa BASIC, o bien si el programa está realizado enteramente en ensamblador, hacer un pequeño cargador (o usar bin2tap o --bastap) y cargar directamente el programa en el Spectrum.

Todo esto se mostró bastante detalladamente en

su momento en el número 1 de nuestro curso.

Con esto, sabemos ensamblar programas creados adecuadamente, de modo que la pregunta es: ¿cómo debo escribir mi programa para que PASMO pueda ensamblarlo? Es sencillo, escribiremos nuestro programa en un fichero de texto con extensión (por ejemplo) .asm. En este fichero de texto se ignorarán las líneas en blanco y los comentarios, que en ASM de Z80 se introducen con el símbolo ";" (punto y coma), de forma que todo lo que el ensamblador encuentre a la derecha de un ; será ignorado (siempre que no forme parte de una cadena). Ese fichero de texto será ensamblado por PASMO y convertido en código binario.

Lo que vamos a ver a continuación son las normas que debe cumplir un programa para poder ser ensamblado en PASMO. Es necesario explicar estas reglas para que el lector pueda consultarlas en el futuro, cuando esté realizando sus propios programas. No te preocupes si no entiendes alguna de las reglas, cuando llegues al momento de implementar tus primeras rutinas, las siguientes normas te serán muy útiles:

- Normas para las instrucciones:
  - Pondremos una sóla instrucción de ensamblador por línea.
  - Como existen diferencias entre los "fin de línea" entre Linux y Windows, es recomendable que los programas se ensamblen con PASMO en la misma plataforma de S.O. en que se han escrito. Si PASMO intenta compilar en Windows un programa ASM escrito en un editor de texto de Linux (con retornos de carro de Linux) es posible que obtengamos errores de compilación (aunque no es seguro). Si os ocurre al compilar los ejemplos que os proporcionamos (están escritos en Linux) y usáis Windows, lo mejor es abrir el fichero .ASM con notepad y grabarlo de nuevo (lo cual lo salvará con formato de retornos de carro de Windows). El fichero "regrabado" con Notepad podrá ser ensamblado sin problemas.
  - Además de una instrucción por línea podremos añadir etiquetas (para referenciar a dicha línea, algo que veremos posteriormente) y comentarios (con ';').
- Normas para los valores numéricos:
  - Todos los valores numéricos se considerarán, por defecto, escritos en decimal.
  - Para introducir valores numéricos en hexadecimal los precederemos del carácter "$", y para escribir valores numéricos en binario lo haremos mediante el carácter "%".
  - Podremos también especificar la base del literal poniéndoles como prefijo las cadena &H ó 0x (para hexadecimal) o &O (para octal).
  - Podemos especificar también los números mediante sufijos: Usando una "H" para hexadecimal, "D" para decimal, "B" para binario u "O" para octal (tanto mayúsculas como minúsculas).
- Normas para cadenas de texto:
  - Podemos separar las cadenas de texto mediante comillas simples o dobles.
  - El texto encerrado entre comillas simples no recibe ninguna interpretación, excepto si se encuentran 2 comillas simples consecutivas, que sirven para introducir una comilla simple en la cadena.
  - El texto encerrado entre comillas dobles permite introducir caracteres especiales al estilo de C/C++ como \n, \r o \t (nueva línea, retorno de carro, tabulador...).
  - El texto encerrado entre comillas dobles también admite \xNN para introducir el carácter correspondiente a un número hexadecimal NN.
  - Una cadena de texto de longitud 1 (un carácter) puede usarse como una constante (valor ASCII del carácter) en expresiones como, por ejemplo, 'C'+10h.
- Normas para los nombres de ficheros:
  - Si vemos que nuestro programa se hace muy largo, podemos partir el fichero en varios ficheros e incluirlos mediante directivas INCLUDE (para incluir ficheros ASM) o INCBIN (para incluir código máquina ya compilado). Al especificar nombres de ficheros, deberán estar entre dobles comillas o simples comillas.
- Normas para los identificadores:
  - Los identificadores son los nombres usados para etiquetas y símbolos definidos mediante EQU y DEFL.
  - Podemos utilizar cualquier cadena de texto, excepto los nombres de las palabras reservadas de ensamblador.
- Normas para las etiquetas:
  - Una etiqueta es un identificador de texto que ponemos poner al principio de cualquier línea de nuestro programa.
  - Podemos añadir el tradicional sufijo ":" a las etiquetas, pero también es posible no incluirlo si queremos compatibilidad con otros

ensambladores que no lo soporten (por si queremos ensamblar nuestro programa con otro ensamblador que no sea pasmo).
    - Para PASMO, cualquier referencia a una etiqueta a lo largo del programa se convierte en una referencia a la posición de memoria donde se ensambla la instrucción o dato donde hemos colocado la etiqueta. Podemos utilizar así etiquetas para hacer referencia a nuestros gráficos, variables, datos, funciones, lugares a donde saltar, etc.
- Directivas:
    - Tenemos a nuestra disposición una serie de directivas para facilitarnos la programación, como DEFB o DB para introducir datos en crudo en nuestro programa, END para finalizar el programa, IF/ELSE/ENDIF en tiempo de compilación, INCLUDE e INCBIN, MACRO y REPT.
    - Una de las directivas más importantes es ORG, que indica la posición origen donde almacenar el código que la sigue. Podemos utilizar diferentes directivas ORG en un mismo programa.
    - Iremos viendo el significado de las directivas conforme las vayamos usando, pero es aconsejable consultar el manual de PASMO para conocer más sobre ellas.
- Operadores:
    - Podemos utilizar los operadores típicos +, -, *. /, así como otros operadores de desplazamiento de bits como >> y <<.
    - Tenemos disponibles operadores de comparación como EQ, NE, LT, LE, GT, GE o los clásicos =, !=, <, >, <=, >=.
    - Existen también operadores lógicos como AND, OR, NOT, o sus variantes &, |, !.
    - Los operadores sólo tienen aplicación en tiempo de ensamblado, es decir, no podemos multiplicar o dividir en tiempo real en nuestro programa usando * o /. Estos operadores están pensados para que podamos poner expresiones como ((32*10)+12), en lugar del valor numérico del resultado, por ejemplo.

ASPECTO DE UN PROGRAMA EN ENSAMBLADOR

Veamos un ejemplo de programa en ensamblador que muestra el uso de algunas de estas normas, para que las podamos entender fácilmente mediante los comentarios incluidos:

```
; Programa de ejemplo para mostrar el aspecto de
; un programa típico en ensamblador para PASMO.
; Copia una serie de bytes a la videomemoria con
; instrucciones simples (sin optimizar).
ORG 40000

valor     EQU  1
destino   EQU  18384

  ; Aquí empieza nuestro programa que copia los
  ; 7 bytes desde la etiqueta "datos" hasta la
  ; videomemoria ([16384] en adelante).

  LD HL, destino     ; HL = destino (VRAM)
  LD DE, datos       ; DE = origen de los datos
  LD B, 6            ; numero de datos a copiar

bucle:               ; etiqueta que usaremos luego

  LD A, (DE)         ; Leemos un dato de [DE]
  ADD A, valor       ; Le sumamos 1 al dato leído
  LD (HL), A         ; Lo grabamos en el destino [HL]
  INC DE             ; Apuntamos al siguiente dato
  INC HL             ; Apuntamos al siguiente destino

  DJNZ bucle         ; Equivale a:
                     ; B = B-1
                     ; if (B>0) goto Bucle
  RET
```

```
datos DEFB 127, %10101010, 0, 128, $FE, %10000000, FFh

END
```

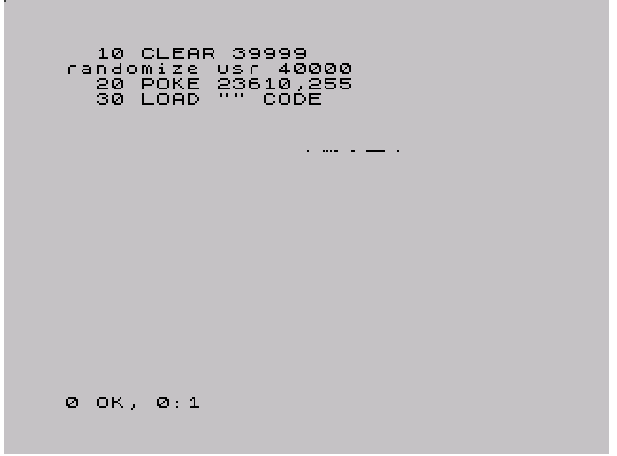
Algunos detalles a tener en cuenta:

- Como veis, se pone una instrucción por línea.
- Los comentarios pueden ir en sus propias líneas, o dentro de líneas de instrucciones (tras ellas).
- Podemos definir "constantes" con EQU para hacer referencia a ellas luego en el código. Son constantes, no variables, es decir, se definen en tiempo de ensamblado y no se cambian con la ejecución del programa. Su uso está pensado para poder escribir código más legible y que podamos cambiar los valores asociados posteriormente de una forma sencilla (es más fácil cambiar el valor asignado en el EQU, que cambiar un valor en todas sus apariciones en el código).
- Podemos poner etiquetas (como "bucle" y "datos" -con o sin dos puntos, son ignorados-) para referenciar a una posición de memoria. Así, la etiqueta "bucle" del programa anterior hace referencia a la posición de memoria donde se ensamblaría la siguiente instrucción que aparece tras ella. Las etiquetas se usan para poder saltar a ellas (en los bucles y condiciones) mediante un nombre en lugar de tener que calcular nosotros la dirección del salto a mano y poner direcciones de memoria. Es más fácil de entender y programar un "DJNZ bucle" que un "DJNZ 40008", por ejemplo. En el caso de la etiqueta "datos", nos permite referenciar la posición en la que empiezan los datos que vamos a copiar.
- Los datos definidos con DEFB pueden estar en cualquier formato numérico, como se ha mostrado en el ejemplo: decimal, binario, hexadecimal tanto con prefijo "$" como con sufijo "h", etc.

Podéis ensamblar el ejemplo anterior mediante:

```
pasmo --tapbas ejemplo.asm ejemplo.tap
```

Una vez cargado y ejecutado el TAP en el emulador de Spectrum, podréis ejecutar el código máquina en BASIC con un "RANDOMIZE USR 40000", y deberéis ver una pantalla como la siguiente:



Salida de ejemplo 1

Los píxeles que aparecen en el centro de la pantalla se corresponden con los valores numéricos que hemos definido en "datos", ya que los hemos copiado desde "datos" hasta la videomemoria. No os preocupéis por ahora si no entendéis alguna de las instrucciones utilizadas, las iremos viendo poco a poco y al final tendremos una visión global y concreta de todas ellas.

LOS REGISTROS

Como ya vimos en la anterior entrega, todo el "trabajo de campo" lo haremos con los registros de la CPU, que no son más que variables de 8 y 16 bits integradas dentro del Z80 y que por tanto son muy rápidos para realizar operaciones con ellos.

El Z80 tiene una serie de registros de 8 bits con nombres específicos:

- A: El Registro A (de 8 bits) es el acumulador. Es un registro que se utiliza generalmente como destino de muchas operaciones aritméticas y de comparaciones y testeos.
- B, C, D, E, H, L: Registros de propósito general, utilizables para gran cantidad de operaciones, almacenamiento de valores, etc.
- I: Registro de interrupción, no lo utilizaremos en nuestros primeros programas. No debemos modificar su valor, aunque en el futuro veremos su uso en las interrupciones del Spectrum.
- R: Registro de Refresco de memoria: lo utiliza internamente la CPU para saber cuándo debe refrescar la RAM. Su valor cambia sólo conforme el Z80 va ejecutando instrucciones, de modo que podemos utilizarlo (leerlo) para obtener valores pseudo-aleatorios entre 0 y 255.

Además, podemos agrupar algunos de estos registros en pares de 16 bits para determinadas operaciones:

- AF: Formado por el registro A como byte más significativo (Byte alto) y por F como byte menos significativo (Byte bajo). Si A vale FFh y B vale 00h, AF valdrá automáticamente "FF00h".
- BC: Agrupación de los registros B y C que se puede utilizar en bucles y para acceder a puertos. También se utiliza como "repetidor" o "contador" en las operaciones de acceso a memoria (LDIR, LDDR, etc.).
- DE, HL: Registros de 16 bits formados por D y E por un lado y H y L por otro. Utilizaremos generalmente estos registros para leer y escribir en memoria en una operación única, así como para las operaciones de acceso a memoria como LDIR, LDDR, etc.

Aparte de estos registros, existen otra serie de registros de 16 bits:

- IX, IY: Dos registros de 16 bits pensados para acceder a memoria de forma indexada. Gracias a estos registros podemos realizar operaciones como: "LD (IX+desplazamiento), VALOR". Este tipo de registros se suele utilizar pues para hacer de índices dentro de tablas o vectores. El desplazamiento es un valor numérico de 8 bits en complemento a 2, lo que nos permite un rango desde -128 a +127 (puede ser negativo para acceder a posiciones de memoria anteriores a IX).
- SP: Puntero de pila, como veremos en su momento apunta a la posición actual de la "cabeza" de la pila.
- PC: Program Counter o Contador de Programa. Como ya vimos en la anterior entrega, contiene la dirección de la instrucción actual a ejecutar. No modificaremos PC directamente moviendo valores a este registro, sino que lo haremos mediante instrucciones de salto (JP, JR, CALL...).

Por último, tenemos disponible un banco alternativo de registros, conocidos como Shadow Registers o Registros Alternativos, que se llaman igual que sus equivalentes principales pero con una comilla simple detrás: A', F', B', C', D'. E', H' y L'.

En cualquier momento podemos intercambiar el valor de los registros A, B, C, D, E, F, H y L con el valor de los registros A', B', C', D', E', F', H' y L' mediante la instrucción de ensamblador "EXX", como veremos más adelante. La utilidad de estos Shadow Registers es almacenar valores temporales y proporcionarnos más registros para operar: podremos intercambiar el valor de los registros actuales con los temporales, realizar operaciones con los registros sin perder los valores originales (que al hacer el EXX se quedarán en los registros Shadow), y después recuperar los valores originales volviendo a ejecutar un EXX.

Ya conocemos los registros disponibles, veamos ahora ejemplos de operaciones típicas que podemos realizar con ellos:

- Meter valores en registros (ya sean valores numéricos directos, de memoria, o de otros registros).
- Incrementar o decrementar los valores de los registros.
- Realizar operaciones (tanto aritméticas como lógicas) entre los registros.
- Acceder a memoria para escribir o leer.

Por ejemplo, las siguientes instrucciones en ensamblador serían válidas:

```
LD C, $00         ; C vale 0
LD B, $01         ; B vale 1
                  ; con esto, BC = $0100
LD A, B           ; A ahora vale 1
LD HL, $1234      ; HL vale $1234 o 4660d
LD A, (HL)        ; A contiene el valor de (4660)
LD A, (16384)     ; A contiene el valor de (16384)
LD (16385), A     ; Escribimos en (16385) el valor de A
ADD A, B          ; Suma: A = A + B
INC B             ; Incrementamos B (B = 1+1 =2)
                  ; Ahora BC vale $0200
INC BC            ; Incrementamos BC
                  ; (BC = $0200+1 = $0201)
```

Dentro del ejemplo anterior queremos destacar el operador "()", que significa "el contenido de la memoria apuntado por". Así, "LD A, (16384)" no quiere decir "mete en A el valor 16384" (cosa que además no se puede hacer porque A es un registro de 8 bits), sino "mete en A el valor de 8 bits que contiene la celdilla de memoria 16384" (equivalente a utilizar en BASIC las funciones PEEK y POKE, como en LET A=PEEK 16384).

Cabe destacar un gran inconveniente del juego de instrucciones del Z80, y es que no es ortogonal. Se dice que el juego de instrucciones de un

microprocesador es ortogonal cuando puedes realizar todas las operaciones sobre todos los registros, sin presentar excepciones. En el caso del Z80 no es así, ya que hay determinadas operaciones que podremos realizar sobre unos registros pero no sobre otros.

Así, si el Z80 fuera ortogonal, podríamos ejecutar cualquiera de estas operaciones:

```
LD BC, 1234h
LD HL, BC
LD SP, BC
EX DE, HL
EX BC, DE
ADD HL, BC
ADD DE, BC
```

Sin embargo, como el Z80 no tiene un juego de instrucciones (J.I. desde este momento) ortogonal, hay instrucciones del ejemplo anterior que no son válidas, es decir, que no tienen dentro de la CPU un microcódigo para que el Z80 sepa qué hacer con ellas:

```
LD SP, BC      ; NO: No se puede cargar el valor un registro en SP,
               ; sólo se puede cargar un valor inmediato NN

EX BC, DE      ; NO: Existe EX DE, HL, pero no EX BC, DE

ADD DE, BC     ; NO: Sólo se puede usar HL como operando destino
               ; en las sumas de 16 bytes con registros de propósito
               ; general.
```

Si el J.I. fuera ortogonal, se podría realizar cualquier operación con cualquier registro, como por ejemplo:

```
LD BC, DE
LD DE, HL
LD SP, BC      ; Se podría realizar
```

Pero "LD SP, BC" es una excepción, y no existe como instrucción del Z80. Y como el caso de "LD SP, BC" existen muchos otros de instrucciones que aceptan unos registros como operandos pero no otros.

La única solución para programar sin tratar de utilizar instrucciones no permitidas es la práctica: con ella acabaremos conociendo qué operaciones podemos realizar y sobre qué registros se pueden aplicar, y realizaremos nuestros programas con estas limitaciones en mente. Iremos viendo las diferentes excepciones caso a caso, pero podemos encontrar las nuestras propias gracias a los errores que nos dará el ensamblador al intentar ensamblar un programa con una instrucción que no existe para el Z80.

No os preocupéis: es sólo una cuestión de práctica. Tras haber realizado varios programas en ensamblador ya conoceréis, prácticamente de memoria, qué instrucciones son válidas para el microprocesador y cuáles no.
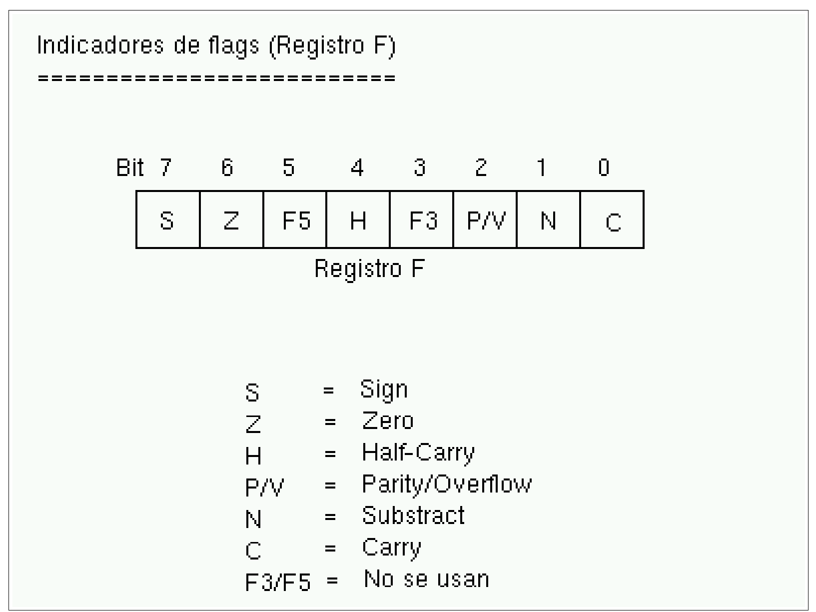
EL REGISTRO DE FLAGS

Hemos hablado del registro de 8 bits F como un registro especial. La particularidad de F es que no es un registro de propósito general donde podamos introducir valores a voluntad, sino que los diferentes bits del registro F tienen un significado propio que cambia automáticamente según el resultado de operaciones anteriores.

Por ejemplo, uno de los bits del registro F, el bit nº 6, es conocido como "Zero Flag", y nos indica si el resultado de la última operación (para determinadas operaciones, como las aritméticas o las de comparación) es cero o no es cero. Si el resultado de la anterior operación resultó cero, este FLAG se pone a uno. Si no resultó cero, el flag se pone a cero.

¿Para qué sirve pues un flag así? Para gran cantidad de tareas, por ejemplo para bucles (repetir X veces una misma tarea poniendo el registro BC al valor X, ejecutando el mismo código hasta que BC sea cero), o para comparaciones (mayor que, menor que, igual que).

Veamos los diferentes registros de flags (bits del registro F) y su utilidad:



Los indicadores de flag del registro F

- Flag S (sign o signo): Este flag se pone a uno si el resultado de la operación realizada en complemento a dos es negativo (es una copia del bit más significativo del resultado). Si por ejemplo realizamos una suma entre 2 números en complemento a dos y el resultado es negativo, este bit se pondrá a uno. Si el resultado es positivo, se pondrá a cero. Es útil para realizar operaciones matemáticas entre múltiples registros: por ejemplo, si nos hacemos una rutina de multiplicación o división de números que permita números negativos, este bit nos puede ser útil en alguna parte de la rutina.
- Flag Z (zero o cero): Este flag se pone a uno si el resultado de la última operación que afecte a los flags es cero. Por ejemplo, si realizamos una operación matemática y el resultado es cero, se pondrá a uno. Este flag es uno de los más útiles, ya que podemos utilizarlo para múltiples tareas. La primera es para los bucles, ya que podremos programar código como:

```
        ; Repetir algo 100 veces
        LD B, 100
bucle:
        (...)        ; código

        DEC B        ; Decrementamos B (B=B-1)
        JR NZ bucle  ; Si el resultado de la operación anterior
                     ; no es cero (NZ = Non Zero), saltar a la
                     ; etiqueta bucle y continuar. DEC B hará
                     ; que el flag Z se ponga a 1 cuando B llegue
                     ; a cero, lo que afectará al JR NZ.
                     ; Como resultado, este trozo de código (..)
                     ; se ejecutará 100 veces.
```

Como veremos en su momento, existe una instrucción equivalente a DEC B + JR NZ que es más cómoda de utilizar y más rápida que estas 2 instrucciones juntas (DJNZ), pero se ha elegido el ejemplo que tenéis arriba para que veáis cómo muchas operaciones (en este caso DEC) afectan a los flags, y la utilidad que estos tienen a la hora de programar.

Además de para bucles, también podemos utilizarlo para comparaciones. Supongamos que queremos hacer en ensamblador una comparación de igualdad, algo como:

```
IF C = B THEN GOTO 1000
ELSE          GOTO 2000
```

Si restamos C y B y el resultado es cero, es que ambos registros contienen el mismo valor:

```
 LD A, C              ; A = C
                      ; Tenemos que hacer esto porque no existe
                      ; una instruccion SUB B, C . Sólo se puede
                      ; restar un registro al registro A.

 SUB B                ; A = A-B
 JP Z, Es_Igual       ; Si A=B la resta es cero y Z=1
 JP NZ, No_Es_Igual   ; Si A<>B la resta no es cero y Z=0
 (...)

Es_Igual:
 (...)
No_Es_Igual:
 (...)
```

- Flag H (Half-carry o Acarreo-BCD): Se pone a uno cuando en operaciones BCD existe un acarreo del bit 3 al bit 4. Es muy probable que no lleguemos a utilizarlo nunca.
- Flag P/V (Parity/Overflow o Paridad/Desbordamiento): En las operaciones que modifican el bit de paridad, este bit vale 1 si el número de unos del resultado de la operación es par, y 0 si es impar. Si, por contra, el resultado de la operación realizada necesita más bits para ser representado de los que nos provee el registro, tendremos un desbordamiento, con este flag a 1. Este mismo bit sirve pues para 2 tareas, y nos indicará una u otra (paridad o desbordamiento) según sea el tipo de operación que hayamos realizado. Por ejemplo, tras una suma, su utilidad será la de indicar el desbordamiento.

  El flag de desbordamiento se activará cuando en determinadas operaciones pasemos de valores 11111111b a 00000000b, por "falta de bits" para representar el resultado o viceversa . Por ejemplo, en el caso de INC y DEC con registros de 8 bits, si pasamos de 0 a 255 o de 255 a 0.
- Flag N (Substract o Resta): Se pone a 1 si la última operación realizada fue una resta. Se utiliza en operaciones aritméticas.
- Flag C (Carry o Acarreo): Este flag se pone a uno si el resultado de la operación anterior no cupo en el registro y necesita un bit extra para ser representado. Este bit es ese bit extra. Veremos su uso cuando tratemos las operaciones aritméticas, en esta misma entrega.

Así pues, resumiendo:

- El registro F es un registro especial cuyo valor no manejamos directamente, sino que cada uno de sus bits tiene un valor especial y está a 1 o a 0 según ciertas condiciones de la última operación realizada que afecte a dicho registro.
- Por ejemplo, si realizamos una operación y el resultado de la misma es cero, se pondrá a 1 el flag de Zero (Z) del registro F, que no es más que su bit número 6.
- No todas las operaciones afectan a los flags, iremos viendo qué operaciones afectan a qué flags conforme avancemos en el curso, en el momento en que se estudia cada instrucción.
- Existen operaciones que se pueden ejecutar con el estado de los flags como condición. Por ejemplo, realizar un salto a una dirección de memoria si un determinado flag está activo, o si no lo está.

INSTRUCCIONES LD

Las operaciones que más utilizaremos en nuestros programas en ensamblador serán sin duda las operaciones de carga o instrucciones LD. Estas operaciones sirven para:

- Meter un valor en un registro.
- Copiar el valor de un registro en otro registro.
- Escribir en memoria (en una dirección determinada) un valor.
- Escribir en memoria (en una dirección determinada) el contenido de un registro.
- Asignarle a un registro el contenido de una dirección de memoria.

La sintaxis de LD en lenguaje ensamblador es:

```
LD DESTINO, ORIGEN
```

Así, gracias a las operaciones LD podemos:

- Asignar a un registro un valor numérico directo de 8 o 16 bits.

```
LD A, 10          ; A = 10
LD B, 200         ; B = 200
LD BC, 12345      ; BC = 12345
```

- Copiar el contenido de un registro a otro registro:

```
LD A, B           ; A = B
LD BC, DE         ; BC = DE
```

- Escribir en posiciones de memoria:

```
LD (12345), A     ; Memoria[12345] = valor en A
LD (HL), 10       ; Memoria[valor de HL] = 10
```

- Leer el contenido de posiciones de memoria:

```
LD A, (12345)     ; A = valor en Memoria[12345]
LD B, (HL)        ; B = valor en Memoria[valor de HL]
```

Nótese cómo el operador () nos permite acceder a memoria. En nuestros ejemplos, LD A, (12345) no significa meter en A el valor 12345 (cosa imposible al ser un registro de 16 bits) sino almacenar en el registro A el valor que hay almacenado en la celdilla número 12345 de la memoria del Spectrum.

En un microprocesador con un juego de instrucciones ortogonal, se podría usar cualquier origen y cualquier destino sin distinción. En el caso del Z80 no es así. El listado completo de operaciones válidas con LD es el siguiente:

Leyenda:

```
N  = valor numérico directo de 8 bits (0-255)
NN = valor numérico directo de 16 bits (0-65535)
r  = registro de 8 bits (A, B, C, D, E, H, L)
rr = registro de 16 bits (BC, DE, HL, SP)
ri = registro índice (IX o IY).
```

Listado:

```
 ; Carga de valores en registros
 LD r, N
 LD rr, NN
 LD ri, NN

 ; Copia de un registro a otro
 LD r, r
 LD rr, rr

 ; Acceso a memoria
 LD r, (HL)
 LD (NN), A
 LD (HL), N
 LD A, (rr)        ; (excepto rr=SP)
 LD (rr), A        ; (excepto rr=SP)
 LD A, (NN)
 LD rr, (NN)
 LD ri, (NN)
 LD (NN), rr
```

```
LD (NN), ri

 ; Acceso indexado a memoria

 ; Acceso indexado a memoria
 LD (ri+N), r
 LD r, (ri+N)
 LD (ri+N), N
```

Además, tenemos una serie de casos "especiales":

```
 ; Manipulación del puntero de pila (SP)
 LD SP, ri
 LD SP, HL

 ; Para manipular el registro I
 LD A, I
 LD I, A
```

Veamos ejemplos válidos y cuál sería el resultado de su ejecución:

```
; Carga de valores en registros
; registro_destino = valor
LD A, 100          ; LD r, N
LD BC, 12345       ; LD rr, NN

; Copia de registros en registros
; registro_destino = registro_origen
LD B, C            ; LD r, r
LD A, B            ; LD r, r
LD BC, DE          ; LD rr, rr

; Acceso a memoria
; (Posicion_memoria) = VALOR o bien
;  Registro = VALOR en (Posicion de memoria)
LD A, (HL)         ; LD r, (rr)
LD (BL), B         ; LD (rr), r
LD (12345), A      ; LD (NN), A
LD A, (HL)         ; LD r, (rr)
LD (DE), A         ; LD (rr), r
LD (BC), 1234h     ; LD (BC), NN
LD (12345), DE     ; LD (NN), rr
LD IX, (12345)     ; LD ri, (NN)
LD (34567), IY     ; LD (NN), ri

; Acceso indexado a memoria
; (Posicion_memoria) = VALOR o VALOR = (Posicion_memoria)
; Donde la posicion es IX+N o IY+N:
LD (IX+10), A      ; LD (ri+N), r
LD A, (IY+100)     ; LD r, (ri+N)
LD (IX-30), 100    ; LD (ri+N), N
```

Haré hincapié de nuevo en el mismo detalle: debido a que el juego de instrucciones del Z80 no es ortogonal, en ocasiones no podemos ejecutar ciertas operaciones que podrían sernos útiles con determinados registros. En ese caso tendremos que buscar una solución mediante los registros y operaciones válidas de que disponemos.

Un detalle muy importante respecto a las instrucciones de carga: en el caso de las operaciones LD, el registro F no ve afectado ninguno de sus indicadores o flags en relación al resultado de la ejecución de las mismas (salvo en el caso de "LD A, I" y "LD A, R").

Esto quiere decir que una operación como "LD A, 0", por ejemplo, no activará el flag de Zero del registro F.

CPU Z80: LOW ENDIAN

Un detalle más sobre nuestra CPU: a la hora de trabajar con datos de 16 bits (por ejemplo, leer o escribir de memoria) conviene tener en cuenta que nuestro Z80 es una CPU del tipo LOW-ENDIAN, es decir, que si almacenamos en la posición de memoria 0000h el valor "1234h", el contenido de las celdillas de memoria sería:

Posición	Valor
0000h	34h
0001h	12h

En otro tipo de procesadores del tipo Big-Endian, los bytes aparecerían escritos en memoria de la siguiente forma:

Posición	Valor
0000h	12h
0001h	34h

Debemos tener en cuenta este dato a la hora de escribir valores de 16 bits en memoria y recuperarlos posteriormente mediante operaciones de acceso a la memoria.

INCREMENTOS Y DECREMENTOS

Entre las operaciones disponibles, tenemos la posibilidad de incrementar (INC) y decrementar (DEC) en 1 unidad el contenido de determinados registros de 8 y 16 bits, así como de posiciones de memoria apuntadas por HL o por IX/IY más un offset (desplazamiento de 8 bits).

Por ejemplo:

```
LD A, 0       ; A = 0
INC A         ; A = A+1 = 1
LD B, A       ; B = A = 1
INC B         ; B = B+1 = 2
INC B         ; B = B+1 = 3
LD  BC, 0
INC BC        ; BC = 0001h
INC B         ; BC = 0101h (ya que B=B+1 y es la parte alta)
DEC A         ; A = A-1 = 0
```

Veamos las operaciones INC y DEC permitidas:

- INC r
- DEC r
- INC rr
- DEC rr

  Donde r puede ser A, B, C, D, E, H o L, y 'rr' puede ser BC, DE, HL, SP, IX o IY. Esta instrucción incrementa o decrementa el valor contenido en el registro especificado.
- INC (HL)
- DEC (HL)

  Incrementa o decrementa el byte que contiene la dirección de memoria apuntada por HL.
- INC (IX+N)
- DEC (IX+N)
- INC (IY+N)
- DEC (IY+N)

  Incrementa o decrementa el byte que contiene la dirección de memoria resultante de sumar el valor del registro IX o el registro IY con un valor numérico de 8 bits en complemento a dos.

Por ejemplo, las siguientes instrucciones serían válidas:

```
INC A           ; A = A+1
DEC B           ; B = B-1
INC DE          ; DE = DE+1
DEC IX          ; IX = IX-1
INC (HL)        ; (HL) = (HL)+1
INC (IX-5)      ; (IX-5) = (IX-5)+1
DEC (IY+100)    ; (IY+100) = (IY+100)+1
```

Unos apuntes sobre la afectación de los flags ante el uso de INC y DEC:

- Si un registro de 8 bits vale 255 (FFh) y lo incrementamos, pasará a valer 0.
- Si un registro de 16 bits vale 65535 (FFFFh)y lo incrementamos, pasará a valer 0.
- Si un registro de 8 bits vale 0 y lo decrementamos, pasará a valer 255 (FFh).
- Si un registro de 16 bits vale 0 (0h) y lo decrementamos, pasará a valer 65535 (FFh).
- En estos desbordamientos no se tomará en cuenta para nada el bit de Carry (acarreo) de los flags (registro F), ni tampoco lo afectarán tras ejecutarse.
- Las operaciones INC y DEC sobre registros de 16 bits (BC, DE, HL, IX, IY, SP) no afectan a los flags. Esto implica que no podemos usar como condición de flag zero para un salto el resultado de instrucciones como "DEC BC", por ejemplo.
- Las operaciones INC y DEC sobre registros de 8 bits y sobre la memoria no afectan al flag de acarreo, pero sí que pueden afectar al flag de Zero (Z), al de Paridad/Overflow (P/V), al de Signo (S) y al de Half-Carry (H).

Lo siguiente que vamos a ver es una tabla de afectación de flags (que podréis ver en muchas tablas de instrucciones del Z80, y a las que conviene que os vayáis acostumbrando). Esta tabla indica cómo afecta cada instrucción a cada uno de los flags:

```
                      Flags
  Instrucción       |S Z H P N C|
----------------------------------
  INC r             |* * * V 0 -|
  INC [HL]          |* * * V 0 -|
  INC [ri+N]        |* * * V 0 -|
  INC rr            |- - - - - -|
  DEC r             |* * * V 1 -|
  DEC rr            |- - - - - -|

Donde:
r = registro de 8 bits
rr = registro de 16 bits (BC, DE, HL, IX, IY)
ri = registro índice (IX, IY)
N = desplazamiento de 8 bits (entre -128 y +127).

Y respecto a los flags:
- = El flag NO se ve afectado por la operación.
* = El flag se ve afectado por la operación acorde al resultado.
0 = El flag se pone a cero.
1 = El flag se pone a uno.
V = El flag se comporta como un flag de Overflow acorde al resultado.
? = El flag toma un valor indeterminado.
```

OPERACIONES MATEMATICAS

Las operaciones aritméticas básicas para nuestro Spectrum son la suma y la resta, tanto con acarreo como sin él. A partir de ellas deberemos crearnos nuestras propias rutinas para multiplicar, dividir, etc.

Suma: ADD (Add)

Nuestro microprocesador Z80 puede realizar sumas de 8 y 16 bits internamente. La instrucción utilizada para ello es "ADD" y el formato es:

```
ADD DESTINO, ORIGEN
```

Las instrucciones disponibles para realizar sumas se reducen a:

```
ADD A, s
ADD HL, ss
ADD ri, rr

Donde:
```

```
s:  Cualquier registro de 8 bits (A, B, C, D, E, H, L),
    cualquier valor inmediato de 8 bits (en el rango 0-255 o -128+127
    en complemento a dos), cualquier dirección de memoria apuntada por HL,
    y cualquier dirección de memoria apuntada por un registro índice con
    desplazamiento de 8 bits.
ss: Cualquier registro de 16 bits de entre los siguientes: BC, DE, HL, SP.
ri: Uno de los 2 registros índices (IX o IY).
rr: Cualquier registro de 16 bits de entre los siguientes excepto el mismo
    registro índice origen: BC, DE, HL, IX, IY, SP.
```

Esto daría la posibilidad de ejecutar cualquiera de las siguientes instrucciones:

```
; ADD A, s
ADD A, B        ; A = A + B
ADD A, 100      ; A = A + 100
ADD A, [HL]     ; A = A + [HL]
ADD A, [IX+10]  ; A = A + [IX+10]

; ADD HL, ss
ADD HL, BC      ; HL = HL + BC
ADD HL, SP      ; HL = HL + SP

; ADD ri, rr
ADD IX, BC      ; IX = IX + BC
ADD IY, DE      ; IY = IY + DE
ADD IY, IX      ; IY = IY + IX
ADD IX, IY      ; IX = IX + IY
```

Por contra, estas instrucciones no serían válidas:

```
ADD B, C      ; Sólo A puede ser destino
ADD BC, DE    ; Sólo puede ser destino HL
ADD IX, IX    ; No podemos sumar un registro índice a él mismo
```

La afectación de los flags ante las operaciones de sumas es la siguiente:

- Para "ADD A, s", el registro N (Substraction) se pone a 0 (lógicamente, ya que sólo se pone a uno cuando se ha realizado una resta). El registro P/V se comporta como un registro de Overflow e indica si ha habido overflow (desbordamiento) en la operación. El resto de flags (Sign, Zero, Half-Carry y Carry) se verán afectados de acuerdo al resultado de la operación de suma.
- Para "ADD HL, ss" y "ADD ri, rr", se pone a 0 el flag N, y sólo se verá afectado el flag de acarreo (C) de acuerdo al resultado de la operación.

O, en forma de tabla de afectación:

```
                         Flags
  Instrucción       |S Z H P N C|
----------------------------------
ADD A, s            |* * * V 0 *|
ADD HL, ss          |- - ? - 0 *|
ADD ri, rr          |- - ? - 0 *|
```

Las sumas realizadas por el Spectrum se hacen a nivel de bits, empezando por el bit de más a la derecha y yendo hacia la izquierda, según las siguientes reglas:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10 (=0 con acarreo)
```

Al sumar el último bit, se actualizará el flag de acarreo si es necesario.

Por ejemplo:

```
      *
  00000100
+ 00000101
 -----------
  00001001

 (* = acarreo de la suma del bit anterior, 1+1=10)
```

Si la suma del último bit (bit 7) requiere un bit extra, se utilizará el Carry Flag del registro F para almacenarlo. Supongamos que ejecutamos el siguiente código:

```
LD A, %10000000
LD B, %10000000
ADD A, B
```

El resultado de la ejecución de esta suma sería: A=128+128=256. Como 256 (100000000b) tiene 9 bits, no podemos representar el resultado con los 8 bits del registro A, de modo que el resultado de la suma sería realmente: A = 00000000 y CarryFlag = 1.

Resta: SUB (Substract)

En el caso de las restas, sólo es posible realizar (de nuevo gracias a la no ortogonalidad del J.I.) la operación "A=A-origen", donde "origen" puede ser cualquier registro de 8 bits, valor inmediato de 8 bits, contenido de la memoria apuntada por [HL], o contenido de la memoria apuntada por un registro índice más un desplazamiento. El formato de la instrucción no requiere 2 operandos, ya que el registro destino sólo puede ser A:

```
SUB ORIGEN
```

Concretamente:

```
SUB r        ; A = A - r
SUB N        ; A = A - N
SUB [HL]     ; A = A - [HL]
SUB [rr+d]   ; A = A - [rr+d]
```

Por ejemplo:

```
SUB B        ; A = A - B
SUB 100      ; A = A - 100
SUB [HL]     ; A = A - [HL]
SUB [IX+10]  ; A = A - [IX+10]
```

Es importante recordar que en una operación "SUB X", la operación realizada es "A=A-X" y no "A=X-A".

Por otra parte, con respecto a la afectación de flags, tenemos la siguiente:

```
Flags:      S Z H P N C
-----------------------
Afectación: * * * V 1 *
```

Es decir, el flag de N (substraction) se pone a 1, para indicar que hemos realizado una resta. El flag de P/V (Parity/Overflow) se convierte en indicar de Overflow y queda afectado por el resultado de la

resta. El resto de flags (Sign, Zero, Half-Carry y Carry) quedarán afectados de acuerdo al resultado de la misma (por ejemplo, si el resultado es Cero, se activará el Flag Z).

Suma con acarreo: ADC (Add with carry)

Sumar con acarreo dos elementos significa realizar la suma de uno con el otro y, posteriormente, sumarle el estado del flag de Carry. Es decir:

```
"ADC A, s"    equivale a    "A = A + s + CarryFlag"
"ADC HL, ss"  equivale a    "HL = HL + ss + CarryFlag"

("s" y "ss" tienen el mismo significado que en ADD y SUB).
```

La tabla de afectación de flags sería la siguiente:

```
                            Flags
 Instrucción            |S Z H P N C|
----------------------------------
ADC A,s                 |* * * V 0 *|
ADC HL,ss               |* * ? V 0 *|
```

La suma con acarreo se utiliza normalmente para sumar las partes altas de palabras de 16 bytes. Se suma la parte baja con ADD y luego la parte alta con ADC para tener en cuenta el acarreo de la suma de la parte baja.

Resta con acarreo: SBC (Substract with carry)

Al igual que en el caso de la suma con acarreo, podemos realizar restas con acarreo, que no son más que realizar una resta de los 2 operandos, tras lo cual restamos además el valor del bit de Carry Flag:

```
"SBC A, s"    equivale a    "A = A - s - CarryFlag"
"SBC HL, ss"  equivale a    "HL = HL - ss - CarryFlag"
```

La tabla de afectación de flags (en este caso con N=1, ya que es una resta):

```
                            Flags
 Instrucción            |S Z H P N C|
----------------------------------
SBC A,s                 |* * * V 1 *|
SBC HL,ss               |* * ? V 1 *|
```

COMPLEMENTO A DOS

A lo largo del presente texto hemos hablado de números en complemento a dos. Complemento a dos es una manera de representar números negativos en nuestros registros de 8 bits, utilizando para ello como signo el bit más significativo (bit 7) del byte.

Si dicho bit está a 0, el número es positivo, y si está a 1 es negativo. Así:

```
01111111    (+127)
01111110    (+126)
01111101    (+125)
01111100    (+124)
(...)
00000100    (+4)
00000011    (+3)
00000010    (+2)
00000001    (+1)
00000000    (0)
11111111    (-1)
11111110    (-2)
11111101    (-3)
11111100    (-4)
```

```
   (...)
10000011    (-125)
10000010    (-126)
10000001    (-127)
10000000    (-128)
```

Podemos averiguar cuál es la versión negativa de cualquier número positivo (y viceversa), invirtiendo el estado de los bits y sumando uno:

```
  +17  = 00010001

  -17 =  11101110   (Invertimos unos y ceros)
      =        +1   (Sumamos 1)
      =  11101111   (-17 en complemento a dos)
```

Como veremos en unos minutos, se eligió este sistema para representar los números negativos para que las operaciones matemáticas estándar funcionaran directamente sobre los números positivos y negativos. ¿Por qué no utilizamos directamente la inversión de los bits para representar los números negativos y estamos sumando además 1 para obtenerlos? Sencillo: si no sumáramos uno y simplemente invirtiéramos los bits, tendríamos 2 ceros (00000000 y 11111111) y además las operaciones matemáticas no cuadrarían (por culpa de los dos ceros). La gracia del complemento a dos es que las sumas y restas binarias lógicas (ADD, ADC, SUB y SBC) funcionan:

```
  Sumemos -17 y 32:

   -17 = 11101111
 + +32 = 00100000
  -----------------
       1 00001111
```

El resultado es 00001111, es decir, 15, ya que 32-17=15. El flag de carry se pone a 1, pero lo podemos ignorar, porque el flag que nos indica realmente el desbordamiento (como veremos a continuación) en operaciones de complemento a dos es el flag de Overflow.

```
  Sumemos ahora +17 y -17:

   +17 = 00010001
 + -17 = 11101111
  ----------------------
       1 00000000
```

Como podéis ver, al sumar +17 y -17 el resultado es 0. Si representáramos los números negativos simplemente como la inversa de los positivos, esto no se podría hacer:

```
   +17 = 00010001
 + -17 = 11101110    <--- (solo bits invertidos)
  ----------------------
       1 11111111    <--- Nos da todo unos, el "cero" alternativo.
```

En complemento a dos, las sumas y restas de números se pueden realizar a nivel lógico mediante las operaciones estándar del Z80. En realidad para el Z80 no hay más que simples operaciones de unos y ceros, y somos nosotros los que interpretamos la información de los operandos y del resultado de una forma que nos permite representar números negativos.

En otras palabras: cuando vemos un uno en el bit más significativo de un resultado, somos nosotros los que tenemos que interpretar si ese bit representa un signo negativo o no: si sabemos que estamos operando con números 0-255, podemos tratarlo como un resultado positivo. Si estábamos

operando con números en complemento a dos, podemos tratarlo como un resultado en complemento a dos. Para el microprocesador, en cambio, no hay más que unos y ceros.

Para acabar, veamos cuál es la diferencia entre el Flag de Carry (C) y el de Overflow (V) a la hora de realizar sumas y restas. El primero (C) se activará cuando se produzca un desbordamiento físico a la hora de sumar o restar 2 números binarios (cuando necesitemos un bit extra para representarlo). El segundo (V), se utilizará cuando se produzca cualquier sobrepasamiento operando con 2 números en complemento a dos.

Como acabamos de ver, en complemento a dos el último bit (el bit 7) nos indica el signo, y cuando operamos con 2 números binarios que nosotros interpretamos como números en complemento a dos no nos basta con el bit de Carry. Es el bit de Overflow el que nos dará información sobre el desbordamiento a un nivel lógico.

En pocas palabras, el bit de Carry se activará si pasamos de 255 a 0 o de 0 a 255 (comportándose como un bit de valor 2 elevado a 8, o 256), y el bit de overflow lo hará si el resultado de una operación en complemento a dos requiere más de 7 bits para ser representado.

Mediante ejemplos:

```
 255+1:

   11111111
+  00000001
  -----------
 1 00000000

   C=1 (porque hace falta un bit extra)
   V=0


 127+1:

   01111111
+  00000001
  -----------
   10000000

   C=0 (no es necesario un bit extra en el registro)
   V=1 (en complemento a dos, no podemos representar +128)
```

En el ejemplo anterior, V se activa porque no ha habido desbordamiento físico (no es necesario un bit extra para representar la operación), pero sí lógico: no podemos representar +128 con 7 bits+signo en complemento a dos.

INSTRUCCIONES DE INTERCAMBIO

Como ya se ha explicado, disponemos de un banco de registros alternativos (los Shadow Registers), y podemos conmutar los valores entre los registros estándar y los alternativos mediante unas determinadas instrucciones del Z80.

El Z80 nos proporciona una serie de registros de propósito general (así como un registro de flags), de nombres A, B, C, D, E, F, H y L. El micro dispone también de unos registros extra (set alternativo conocido como Shadow Registers) de nombre A', B', C', D', E', F', H' y L', que aprovecharemos en cualquier momento de nuestro programa. No obstante, no podremos hacer uso directo de estos registros en instrucciones en ensamblador. No es posible, por ejemplo, ninguna de las siguientes instrucciones:

```
LD B', $10
INC A'
LD HL', $1234
LD A', ($1234)
```

La manera de utilizar estos registros alternativos es conmutar sus valores con los registros estándar mediante la instrucción "EXX", cuyo resultado es el intercambio de B por B', C por C', D por D', E por E', H por H' y L por L'. Supongamos que tenemos los siguientes valores en los registros:

Registro	Valor	Registro	Valor
B	A0h	B'	00h
C	55h	C'	00h
D	01h	D'	00h
E	FFh	E'	00h
H	00h	H'	00h
L	31h	L'	00h

En el momento en que realicemos un EXX, los registros cambiarán de valor por la "conmutación" de bancos:

Registro	Valor	Registro	Valor
B	00h	B'	A0h
C	00h	C'	55h
D	00h	D'	01h
E	00h	E'	FFh
H	00h	H'	00h
L	00h	L'	31h

Si realizamos de nuevo EXX, volveremos a dejar los valores de los registros en sus "posiciones" originales. EXX (mnemónico ensamblador derivado de EXchange), simplemente intercambia los valores entre ambos bancos.

Aparte de la instrucción EXX, disponemos de una instrucción EX AF, AF' , que, como el lector imagina, intercambia los valores de los registros AF y AF'. Así, pasaríamos de:

Registro	Valor	Registro	Valor
A	01h	A'	00h
F	10h	F'	00h

a:

Registro	Valor	Registro	Valor
A	00h	A'	01h
F	00h	F'	10h

Realizando de nuevo un EX AF, AF' volveríamos a los valores originales en ambos registros.

De esta forma podemos disponer de un set de registros extra con los que trabajar. Por ejemplo, supongamos que programamos una porción de código donde queremos hacer una serie de cálculos entre registros y después dejar el resultado en una posición de memoria, pero no queremos perder los valores actuales de los registros (ni tampoco hacer uso de la pila, que veremos en su momento). En ese caso, podemos hacer:

```
; Una rutina a la que saltaremos gracias a la
; etiqueta que definimos aquí:
MiRutina:

    ; Cambiamos de banco de registros:
    EXX
    EX AF, AF'

    ; Hacemos nuestras operaciones
    LD A, (1234h)
    LD B, A
    LD A, (1235h)
    INC A
    ADD A, B
    ; (...etc...)
    ; (...aquí más operaciones...)

    ; Grabamos el resultado en memoria
    LD (1236h), A

    ; Recuperamos los valores de los registros
    EX AF, AF'
    EXX

    ; Volvemos al lugar de llamada de la rutina
    RET
```

Aparte de EXX y EX AF, AF' tenemos disponibles 3 instrucciones de intercambio más que no trabajan con los registros alternativos, sino entre la memoria y registros, y la pila (o memoria en general) y los registros HL, IX e IY.

Instrucción	Resultado
EX DE, HL	Intercambiar los valores de DE y HL.
EX (SP), HL	Intercambiar el valor de HL con el valor de 16 bits de la posición de memoria apuntada por el registro SP (por ejemplo, para intercambiar el valor de HL con el del último registro que hayamos introducido en la pila).
EX (SP), IX	Igual que el anterior, pero con IX.
EX (SP), IY	Igual que el anterior, pero con IY.

La primera de estas instrucciones nos puede ser muy útil en nuestros programas en ensamblador, ya que nos permite intercambiar los valores de los registros DE y HL. Las 3 instrucciones restantes permiten intercambiar el valor apuntado por SP (en memoria) por el valor de los registros HL, IX o IY. Como ya hemos comentado cuando hablamos del carácter Low-Endian de nuestra CPU, al escribir en memoria (también en la pila) primero se escribe el Byte Bajo y luego el Byte Alto. Posteriormente lo leeremos de la misma forma, de tal modo que si los bytes apuntados en la pila (en memoria) son "FF 00h", al hacer el EX (SP), HL, el registro HL valdrá "00FFh".

Nótese que aprovechando la pila (como veremos en su momento) también podemos intercambiar los valores de los registros mediante:

```
PUSH BC
PUSH DE
POP BC
POP DE
```

Si queréis comprobarlo, podéis hacerlo mediante el siguiente programa:

```
; Ejemplo que muestra el intercambio de registros
; mediante el uso de la pila (PUSH/POP).
ORG 40000

; Cargamos en DE el valor 12345 y
; realizamos un intercambio de valores
; con BC, mediante la pila:
LD DE, 12345
LD BC, 0

PUSH DE
PUSH BC
POP DE
POP BC

; Volvemos, ahora BC=DE y DE=BC
RET
```

Lo ensamblamos:

```
pasmo --tapbas cambio.asm cambio.tap
```

Tras esto lo cargamos en un emulador de Spectrum (como un fichero TAP), nos vamos al BASIC y tecleamos "PRINT AT 10, 10; USR 40000". En pantalla aparecerá el valor "12345", ya que las rutinas llamadas desde BASIC devuelven sus resultados en BC, y nosotros hemos hecho un intercambio mediante la pila, entre DE y BC.

En su momento veremos cómo funciona la pila, por ahora basta con saber que tenemos la posibilidad de intercambiar registros mediante el uso de la misma. Podríamos haber optado por no explicar este pequeño truco hasta haber hablado de la pila, pero nos parece más conveniente el hecho de tener toda la información sobre ensamblador agrupada de forma que cuando en el futuro (una vez terminado el curso completo) busquéis información sobre instrucciones para intercambiar valores de registros, podáis

encontrarla toda junta, como un libro o una guía de referencia. Como hemos comentado al principio de esta entrega, resulta muy complicado explicar un lenguaje tan interrelacionado de forma que no se solapen diferentes áreas, de modo que la comprensión total de muchos de los conceptos se alcanzará con una segunda lectura del curso completo.

EN RESUMEN

En esta entrega hemos visto la sintaxis de los programas en ensamblador (o, al menos, la sintaxis general de PASMO, el ensamblador que recomendamos), así como una descripción completa del juego de registros del Z80, incluyendo entre ellos el registro de flags F.

Además, hemos comenzado a ver nuestras primeras instrucciones del lenguaje ensamblador, en especial las instrucciones de carga, incremento y decremento, y aritméticas.

En la próxima entrega continuaremos detallando las diferentes instrucciones del Z80, ejemplos de uso y su efecto sobre los flags del registro F. Hasta entonces, os recomiendo la lectura del fichero "progcard.txt" adjunto con este artículo, donde encontraréis una gran referencia de instrucciones y flags.

SROMERO

FICHEROS

Ejemplo de programa en ASM
Fichero tap del ejemplo ejemplo.asm
Referencia del juego de instrucciones del Z80 y cómo su ejecución afecta a los flags
Programa en ASM que muestra el uso de la pila para el intercambio de registros
Fichero tap del ejemplo cambio.asm

Juego de caracteres
Web del Z80
Z80 Reference de WOS
Z80 Reference de TI86
Microfichas de CM de MicroHobby
Tablas de ensamblado y t-estados (pulsar en z80.txt, z80_reference.txt, z80time.txt)
Z80 Reference de WOS
Curso de ensamblador de z80.info
Pasmo




ZX
magazine